TAYPE: A Policy-Agnostic Language for Oblivious Computation

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

Secure multiparty computation (MPC) allows for joint computation over private data from multiple entities, usually backed by powerful cryptographic techniques that protect sensitive data. Several high-level programming languages have been proposed to make writing MPC applications accessible to non-experts. These languages typically require developers to enforce security policies within the logic of the secure application itself, making it difficult to update security requirements, or to experiment with different policies. This paper presents the design and implementation of TAYPE, a language that permits security concerns to be decoupled from the program logic. To do so, TAYPE provides the first implementation of *oblivious algebraic data types* and *tape semantics*, two language features recently proposed by a core calculus for oblivious computation, $\lambda_{\text{OADT}✚}$. We evaluate our implementation of TAYPE on a range of benchmarks, demonstrating its ability to encode a range of security polices for a rich class of data types.

147

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 type systems, Algebraic Data Types

ACM Reference Format:
Qianchuan Ye and Benjamin Delaware. 2023. TAYPE: A Policy-Agnostic Language for Oblivious Computation. *Proc. ACM Program. Lang.* 7, PLDI, Article 147 (June 2023), 25 pages. https://doi.org/10.1145/3591261

1 INTRODUCTION

Secure multiparty computation (MPC) allows multiple parties to perform a joint computation while keeping their sensitive data private. This enables, for example, a group of hospitals to calculate statistics about the populations they serve without directly sharing patient records with each other. Since its formal introduction by Yao [1982], secure multiparty computation has found many privacy-focused applications, including secure auction, voting and privacy-preserving machine learning [Evans et al. 2018; Hastings et al. 2019; Laud and Kamm 2015]. Core to these techniques are protocols that use powerful cryptographic operations to secure private data. In this setting, there is often a fundamental tradeoff between privacy and performance, as the more information that must be hidden, the more computation is needed to hide it. If one of the aforementioned hospitals chooses to release all of its patients' records, for example, this computation becomes quite cheap![1]

Several high-level programming languages have been developed to help non-experts in cryptography write and deploy applications built on these protocols [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; Zahur and Evans 2015]. Unfortunately, all of these languages rely on the program (and thus the programmer) to enforce the desired privacy policy. As a simple example, consider the privacy policy where *either*

[1]Of course, the legal bills incurred by such a disclosure may limit the benefit to the hospital's bottom line.

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

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/6-ART147
https://doi.org/10.1145/3591261

the personally identifiable information (PII) in a patient record *or* its medical data (but not both) can be accessed in an insecure way, as is legally required by the Health Insurance Portability and Accountability Act (HIPAA). This policy does not simply attach a private or public designation to each field, but it instead stipulates a relation between how its fields are accessed. When using the aforementioned languages, a programmer has to explicitly enforce this policy by performing appropriate secure operations needed to maintain this relationship at each point a medical record is accessed, thus embedding the policy into the logic of the program itself. As the policies become more complex, so does the logic required to enforce them, particularly in the case of recursive data types that may hide their structure, e.g., private decision trees. The entanglement of privacy and computational concerns makes it hard to update a program to satisfy new privacy requirements, and to explore the aforementioned tradeoffs between performance and privacy guarantees.

Recently, Ye and Delaware [2022] proposed a core calculus, $\lambda_{\text{OADT}\maltese}$, which permitted security concerns to be decoupled from the program logic of a multiparty computation. The first key component of their solution was a dependent security type system in which *oblivious algebraic data types* could be encoded. These types equip private data with a *public view*, and guarantee that every private value with the same view is indistinguishable, i.e., an attacker can learn nothing about private data other than what its public view entails. The second key component was a novel *tape semantics* that uses security information provided by the type system to dynamically *repair* any potential information leaks at runtime. These components allow the programmer to write a program as normal, and then combine it with the desired public view, relying on the tape semantics to patch any information leaks. Unfortunately, $\lambda_{\text{OADT}\maltese}$ lacked an accompanying implementation.

This paper presents a programming language for writing secure multiparty computations, Taype, that implements both the oblivious algebraic data types and tape semantics proposed in $\lambda_{\text{OADT}\maltese}$. Taype is equipped with a bidirectional type checker that enforces correct use of secure operations, and automatically infers annotations that enable potential leaks to be repaired. Our implementation is realized in a compilation pipeline, shown in Figure 1, that translates a Taype program and privacy policy (in the form of a public view) to an OCaml implementation which, when linked with a cryptographic backend, can be used by client programs to securely compute functions over private data. The main challenge that this toolchain must overcome is how to securely implement these two language features in a standard functional programming language. To implement oblivious types, our key idea is to represent dependently typed oblivious data using an *oblivious array*, and the types themselves as sizes indexing into an array. To implement the tape semantics, we equip each type, including function types, with a *leaky structure* that reifies potentially leaky operations into a distinguished data type and inserts repairs when values of this type are used.

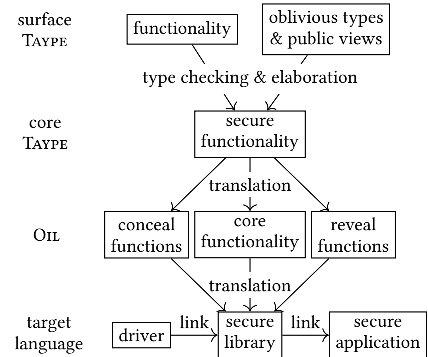


Fig. 1. Compilation pipeline

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

- We implement a bidirectional type checker for an extension of $\lambda_{\text{OADT}\maltese}$. Given a source program, this checker outputs a fully-annotated version in a typed core language called core Taype.
- We present a translation from core Taype to Oil, a ML-style functional language with rank-1 polymorphism, built-in oblivious arrays, and secure array operations. In addition to translating the core functionality of the application, our translation also produces routines for concealing and revealing private data, which clients need to build a complete MPC application.

- We evaluate our implementation against several case studies and micro-benchmarks. Our experiments feature a diverse set of computations and a range of security policies, including the aforementioned medical record example, and also demonstrate that tradeoffs between privacy and performance can be made easily with our approach.

2 BACKGROUND AND OVERVIEW

To demonstrate our approach, consider a simple list membership predicate written in TAYPE, `elem`, shown in Figure 2. Suppose Alice, the owner of a list, and Bob, the owner of an integer, want to check if Bob's integer occurs in Alice's list, without revealing any information beyond their own input and the result. We adopt a standard semi-honest threat model, where all parties can observe the whole execution trace generated by a small-step semantics [Ye and Delaware 2022].

```
data list = Nil | Cons ℤ list
fn elem : ℤ → list → 𝔹 =
  λy xs ⇒
    case xs of Nil ⇒ False
    | Cons x xs′ ⇒
      if x ≡ y then True
      else elem y xs′
```

Fig. 2. List membership predicate

Under this threat model, if `elem` is executed naively, Alice and Bob can glean private information from both the shape of Alice's list and the control flow of the program. For instance, assume Alice has the list `Cons [1] (Cons [2] Nil)`, and Bob's integer is `[1]`, where the square brackets denote *oblivious values* which only the data owner can directly observe. Just from inspecting the public constructors of Alice's list, Bob can infer that it contains two elements. Additionally, by examining the execution trace of `elem`, Alice learns that Bob's integer is 1, and Bob learns that the first element of Alice's private list must also be 1, because `elem` returns after the first comparison. Note that both parties have to release *some* information in order for `elem` to terminate, as the number of intermediate steps in the execution inevitably reveals an upper bound on the number of elements in the list. Alice may be okay with sharing the size of the list, but not its elements, or with releasing some upper bound on its length, for example. In the latter case, if Bob and Alice agree that the length of the list will be no more than 5, for example, Bob should not learn the exact length of Alice's list by knowing how `elem` computes. TAYPE provides exactly this guarantee: participants choose what public information to share as part of the security policy, and its type system ensures that all public data and computation only depend on this *public view*.

2.1 Oblivious Programs

Oblivious Algebraic Data Types. We define an oblivious data type in TAYPE as a dependent type that takes its public view as an argument. Figure 3 shows the definition of an oblivious list, $\widehat{\texttt{list}}$, whose public view `k` is its maximum length. In general, a public view can be any data type. By convention, we use $\hat{\cdot}$ to denote the secure versions of types and functions. The body of $\widehat{\texttt{list}}$ is built up from a set of *oblivious type formers*, e.g., $\widehat{\mathbb{Z}}$ and $\widehat{+}$ are type formers for oblivious fixed-width integers and oblivious sum, respectively. The key idea behind oblivious ADTs is that the representation of private data is stipulated by its public view. As a consequence, private data with the same public view are *indistinguishable* to an attacker. For example, all oblivious lists with a maximum length of two have the same representation: $\widehat{\texttt{list}}\ 2 \equiv \mathbb{1} \mathbin{\widehat{+}} \widehat{\mathbb{Z}} \mathbin{\widehat{\times}} (\mathbb{1} \mathbin{\widehat{+}} \widehat{\mathbb{Z}} \mathbin{\widehat{\times}} \mathbb{1})$.

```
obliv list̂ (k : ℤ) =
  if k ≡ 0 then 𝟙
  else 𝟙 +̂ ℤ̂ ×̂ list̂ (k−1)

// Id, age, height and weight
data patient = Patient ℤ ℤ ℤ ℤ
data patient_view = Known_id ℤ
                  | Known_data ℤ ℤ
obliv patient̂ (v : patient_view) =
  case v of
  | Known_id _ ⇒ ℤ̂ ×̂ ℤ̂ ×̂ ℤ̂
  | Known_data _ _ ⇒ ℤ̂ ×̂ ℤ̂
```

Fig. 3. Example oblivious ADTs

In this example, the oblivious type $\widehat{\texttt{list}}\ 2$ is computed to the *oblivious type value* on the right hand side. Intuitively, every oblivious list of this type is padded to length 2, even if its actual length

may be shorter than 2, to avoid leaking structural information, thanks to the use of oblivious sum $\widehat{+}$. An adversary can not distinguish between a left injection and a right injection of an oblivious sum by inspecting their "tags" or their payload. In the case of `list`, this means the attacker can not tell `Nil` and `Cons` apart.

Figure 3 also presents an oblivious type for a simplified version of the medical record with the "either-or" policy from the introduction. A `patient` record consists of their ID, age, height and weight. The public view in this example consists of either a patient's ID (`Known_id`), or their height and weight (`Known_data`); their age is always private. The corresponding oblivious type $\widehat{\texttt{patient}}$ is straightforward: it is the oblivious data that has been omitted from the public view. If the ID is disclosed, for example, then the oblivious type is essentially an encrypted version of the remaining 3 fields. Oblivious ADTs are expressive enough to directly support this kind of "either-or" policy.

To ensure no private information leaks through control flow channels, Taype includes a set of oblivious operations. One such operation is the special conditional `mux`, which returns an oblivious value according to its private condition, as illustrated in the following execution trace:

$$\texttt{mux ([3]} \mathrel{\widehat{\leq}} \texttt{[4]) ([5]} \mathrel{\widehat{+}} \texttt{[1]) ([6]} \mathrel{\widehat{+}} \texttt{[1])} \longrightarrow \texttt{mux [True] ([5]} \mathrel{\widehat{+}} \texttt{[1]) ([6]} \mathrel{\widehat{+}} \texttt{[1])}$$
$$\longrightarrow \texttt{mux [True] [6] ([6]} \mathrel{\widehat{+}} \texttt{[1])} \longrightarrow \texttt{mux [True] [6] [7]} \longrightarrow \texttt{[6]}$$

We abuse the notation $\widehat{+}$ to also denote the secure addition of oblivious integers. Unlike the standard `if`, `mux` fully evaluates *both* branches before taking the last step to the final oblivious result. As a consequence, it always generates the same execution trace regardless of the value of the private condition: replacing `[3]` with `[5]` in the condition yields the same execution trace:

$$\texttt{mux ([5]} \mathrel{\widehat{\leq}} \texttt{[4]) ([5]} \mathrel{\widehat{+}} \texttt{[1]) ([6]} \mathrel{\widehat{+}} \texttt{[1])} \longrightarrow \texttt{mux [False] ([5]} \mathrel{\widehat{+}} \texttt{[1]) ([6]} \mathrel{\widehat{+}} \texttt{[1])}$$
$$\longrightarrow \texttt{mux [False] [6] ([6]} \mathrel{\widehat{+}} \texttt{[1])} \longrightarrow \texttt{mux [False] [6] [7]} \longrightarrow \texttt{[7]}$$

As the attacker can not observe the values of private integers at each step, nothing can be inferred from inspecting the execution trace.

With oblivious types and these oblivious operations, it is already possible to implement a secure `elem` function. Writing oblivious programs is nontrivial, however, especially for more complex data structures. Moreover, this strategy entwines the security policy with the program logic: the programmer needs to track what information is private and insert appropriate secure operations, e.g., `mux`, and restructure the program control flow to depend only on the public inputs. Using this approach, Alice and Bob would have to write a version of `elem` specialized to their desired policy.

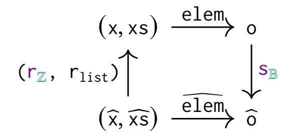


Fig. 4. Recipe of a secure `elem` function

A modular approach. Instead of writing an oblivious program for each public view, Ye and Delaware [2022] proposed a more modular paradigm for writing oblivious programs, one that decouples security and programmatic concerns. The key idea behind their approach is to define a single `elem` function, and to construct the corresponding secure version for a particular view using the "recipe" shown in Figure 4. To compute an oblivious value, this recipe first converts private inputs to public versions, then applies `elem` to the "revealed" arguments, and finally converts the public output back to the oblivious result. From the cryptographic perspective, the arrow labeled $(\texttt{r}_{\mathbb{Z}},\ \texttt{r}_{\texttt{list}})$ corresponds to decrypting the private inputs, while the arrow labeled $\texttt{s}_{\mathbb{B}}$ corresponds to encrypting the output. We call these two conversion functions *section* and *retraction*. As the names suggest, retraction is the left inverse for section: decrypting an encrypted value should result in the original value. Figure 5 presents a secure implementation of $\widehat{\texttt{elem}}$ using this recipe. Notably, the security concerns are delegated to the section and retraction functions, and `mux` is nowhere to be seen! Unfortunately, this naive implementation is thoroughly insecure, as the private input is completely leaked by the initial retraction. However, as we will see shortly, the security guarantees can be recovered by a special kind of *tape semantics* that repairs potential leaks at runtime.

```
#[section]
fn s_list : (k : ℤ) → list → list^ k = λk xs ⇒
  if k ≡ 0 then ()
  else tape (case xs of Nil ⇒ inl^ ()
            | Cons x xs′ ⇒
              inr^ ⟨tape (s_ℤ x), s_list (k−1) xs′⟩)
```

```
#[retraction]
fn r_list : (k : ℤ) → list^ k → list = λk ⇒
  if k ≡ 0 then λ_ ⇒ Nil
  else λxs ⇒ case^ xs of inl^ _ ⇒ Nil
             | inr^ ⟨x, xs′⟩ ⇒
               Cons (r_ℤ x) (r_list (k−1) xs′)
```

```
#[safe] fn elem^ : (k : ℤ) → ℤ^ → list^ k → 𝔹^ = λk x xs ⇒ tape (s_𝔹 (elem (r_ℤ x) (r_list k xs)))
```

Fig. 5. An oblivious implementation of $\widehat{\texttt{elem}}$

This strategy enjoys multiple benefits. First, the core program logic is easier to write and reason about, because it is simply a normal functional program, just like `elem`. Second, these core functions are agnostic to a particular security policy. To share the exact length of the list, we only need to choose a different r_{list} and s_{list}; `elem` itself remains unchanged. This frees users from writing different versions of the same function for different security policies. Third, this approach allows users to experiment and trade off between performance and security guarantee. Sharing the exact length of the list will result in better performance, for example, if both parties agree to this policy.

Tape semantics. The key idea behind the tape semantics is to repair potentially leaky expressions at runtime. To understand how, consider the following execution trace of a simple TAYPE expression.

$$\begin{aligned}&\texttt{tape}\ (s_{\mathbb{Z}}\ (\widehat{\texttt{if}}\ [\texttt{True}]\ \texttt{then}\ 3\ \texttt{else}\ 4)) \longrightarrow \texttt{tape}\ (\widehat{\texttt{if}}\ [\texttt{True}]\ \texttt{then}\ s_{\mathbb{Z}}\ 3\ \texttt{else}\ s_{\mathbb{Z}}\ 4)\\ &\longrightarrow^{*} \texttt{tape}\ (\widehat{\texttt{if}}\ [\texttt{True}]\ \texttt{then}\ [3]\ \texttt{else}\ [4]) \longrightarrow \texttt{mux}\ [\texttt{True}]\ [3]\ [4] \longrightarrow [3]\end{aligned} \tag{1}$$

The new conditional $\widehat{\texttt{if}}$ is a *leaky operation*: it takes a private condition, similar to `mux`, but can have non-oblivious branches. In this example, the branches are public booleans, but in general they can be anything (including functions). Since the private condition would be leaked if we execute this $\widehat{\texttt{if}}$ expression using the semantics of `mux`, we simply choose not to. Instead, we defer reducing $\widehat{\texttt{if}}$ until its branches become oblivious. This example makes progress by distributing the surrounding $s_{\mathbb{Z}}$ into both branches and evaluating them to oblivious values. The outermost `tape` operation ensures the expression will be *eventually oblivious*, and securely reduces the whole expression into a `mux`, once both branches are oblivious values. Note that swapping the private condition `[True]` with `[False]` in this example produces the exact same trace, modulo oblivious values.

Another leaky operation is the primitive integer retraction $r_{\mathbb{Z}}$:

$$\begin{aligned}&\texttt{tape}\ (s_{\mathbb{Z}}\ (r_{\mathbb{Z}}\ [3] + r_{\mathbb{Z}}\ [2])) \longrightarrow \texttt{tape}\ (s_{\mathbb{Z}}\ (r_{\mathbb{Z}}\ ([3]\ \widehat{+}\ [2])))\\ &\longrightarrow \texttt{tape}\ (s_{\mathbb{Z}}\ (r_{\mathbb{Z}}\ [5])) \longrightarrow \texttt{tape}\ [5] \longrightarrow [5]\end{aligned} \tag{2}$$

Similar to $\widehat{\texttt{if}}$, the evaluation of $r_{\mathbb{Z}}$ is also deferred. This example progresses by distributing the addition operator into retraction, and securely adding each oblivious operand. The potential leak introduced by $r_{\mathbb{Z}}$ is eventually patched by $s_{\mathbb{Z}}$, which "cancels" the retraction when they meet. Since `[5]` is already an oblivious value, `tape` becomes a no-op in this example.

Figure 5 shows the full implementation of the section and retraction functions of $\widehat{\texttt{list}}$. In r_{list}, we use the leaky $\widehat{\texttt{case}}$ operation to eliminate the oblivious sum `xs`, using a similar idea to $\widehat{\texttt{if}}$ (Section 3).

To see how recursive functions are unrolled, consider an execution trace of $\widehat{\texttt{elem}}$ of the form:[2]

$$\begin{aligned}&\texttt{elem}\ (r_{\mathbb{Z}}\ [3])\ (r_{list}\ 2\ (\widehat{\texttt{inr}}\ \langle[0],\ \widehat{\texttt{inl}}\ ()\rangle))\\ &\longrightarrow^{*} \texttt{elem}\ (r_{\mathbb{Z}}\ [3])\ (\widehat{\texttt{if}}\ [\texttt{False}]\ \texttt{Nil}\ (\texttt{Cons}\ (r_{\mathbb{Z}}\ [0])\ (r_{list}\ 1\ (\widehat{\texttt{inl}}\ ()))))\\ &\longrightarrow^{*} \widehat{\texttt{if}}\ [\texttt{False}]\ (\texttt{case}\ \texttt{Nil}\ \texttt{of}\ \ldots)\ (\texttt{case}\ \texttt{Cons}\ (r_{\mathbb{Z}}\ [0])\ (r_{list}\ 1\ (\widehat{\texttt{inl}}\ ()))\ \texttt{of}\ \ldots)\\ &\longrightarrow^{*} \widehat{\texttt{if}}\ [\texttt{False}]\ \texttt{False}\ (\texttt{if}\ r_{\mathbb{Z}}\ [0] \equiv r_{\mathbb{Z}}\ [3]\ \texttt{then}\ \texttt{True}\ \texttt{else}\ \texttt{elem}\ (r_{\mathbb{Z}}\ [3])\ (r_{list}\ 1\ (\widehat{\texttt{inl}}\ ())))\end{aligned}$$

Retracting the (secure) list argument of $\widehat{\texttt{elem}}$ (i.e., the section of `Cons 0 Nil`) with public view `2` results in the appearance of a leaky $\widehat{\texttt{if}}$ "guarding" the corresponding argument to `elem`. Since

[2] In the actual trace, the highlighted expression is fully evaluated; we leave it unevaluated here to keep the example compact.

```
fn elem_c :⊤ ℤ_⊤ → list_⊤ → 𝔹 =
  λ(y :⊤ ℤ) (xs :⊤ list) ⇒
    case xs of Nil ⇒ ↑False
    | Cons x xs′ ⇒ if x ≡ y then ↑True
                    else elem_c y xs′
```

```
fn elem_o : ℤ̃ → list̃ → 𝔹̃ =
  λy xs ⇒
    câse_list îf_𝔹 xs (prom_𝔹 False)
      (λx xs′ ⇒ îf îf_𝔹 (x ≡̃ y) (prom_𝔹 True)
                      (elem_o y xs′))
```

Fig. 6. A fully annotated implementation of `elem` in core Taype ($\texttt{elem}_c$) and its translation in Oil ($\texttt{elem}_o$)

the branches of this $\widehat{\texttt{if}}$ are not oblivious values, the tape semantics distributes the case expression of `elem` to each branch. Reducing both case statements eventually results in a recursive call, `elem` ($r_{\mathbb{Z}}$ `[3]`) ($r_{\texttt{list}}$ `1` ($\widehat{\texttt{inl}}$ `()`)). Note that the recursive argument is a retraction with public view `1`, i.e. `elem` has been automatically restructured to recurse on the public view of its recursive argument, since that is how $r_{\texttt{list}}$ is defined. In general, the unrolling of recursive functions follows the unrolling of the retraction function used in our recipe.

While Ye and Delaware [2022] formalized a core calculus of oblivious algebraic data types and tape semantics, $\lambda_{\text{OADT}✚}$, it lacked both an algorithmic type checker and implementation; this paper presents the design and implementation of a language for oblivious computation with both.

2.2 Type checking and core Taype

The input to our compiler is a program written in *surface Taype*, such as `elem`, $r_{\texttt{list}}$, and $s_{\texttt{list}}$ from Figures 2 and 5. This language is equipped a *bidirectional type checker* [Dunfield and Krishnaswami 2021] that enforces correct use of secure and leaky operations, which ensures that all well-typed programs are oblivious. After type checking, programs in this language are elaborated into an intermediate language called *core Taype* (Section 3).

Core Taype programs are fully annotated with types and, crucially, *leakage labels*. Leakage labels track whether an expression contains potential leaks, i.e. whether it contains any leaky operations: we say an expression is *leaky* (labelled $\top$) if so, and *safe* (labelled $\bot$) otherwise. For example, $\widehat{\texttt{if}}$ `x then 1 else 2` is obviously leaky, as executing it naively leaks the private condition, while `mux x then [1] else [2]` and `False` are safe. In contrast to $\lambda_{\text{OADT}✚}$, an addition in core Taype is its *promotion* operation $\uparrow$, which explicitly casts a safe expression to a leaky one, to help with the translation: the $\uparrow$`False` on the third line of $\texttt{elem}_c$ in Figure 6 is treated as a leaky expression, for example. Note that none of the label annotations or promotion are required in the surface language: they are either inferred or automatically inserted during elaboration.

2.3 Translating to Oil

The next compilation phase translates programs in core Taype into Oil, the *OADT intermediate language* (Section 4). Oil is an ML-style functional language with rank-1 polymorphism, extended with an *oblivious array* and its operations. These oblivious “primitives” will eventually be implemented by a cryptographic backend in the target language (Section 5), and Oil is agnostic to the particular implementation. Oil is designed to be a common subset of most standard functional languages, so that translating Oil to a particular language, e.g., OCaml, is straightforward. The main challenge in this phase is expressing the unique features of Taype that do not appear in conventional languages, particularly oblivious types (i.e. dependent types) and its tape semantics. Many of the core ingredients of this translation can be seen in Figure 6, which gives the fully elaborated implementation of core Taype and its corresponding Oil version.

Translating oblivious type definitions. As Oil does not have type-level computation, the definition of the oblivious type $\widehat{\texttt{list}}$ is translated into a function from the public view to its *size* $\mathbb{N}$, shown in Figure 7. As we will see shortly, the size of an oblivious type can be used to access secure data residing in an oblivious array.

```
data 𝒜̃ = prom_𝒜 𝒜 | if̂_𝒜 𝒜 𝒜̃ 𝒜̃
data ℤ̃ = r_ℤ 𝒜 | prom_ℤ ℤ | if̂_ℤ 𝒜 ℤ̃ ℤ̃
fn s̃_ℤ : ℤ̃ → 𝒜̃ = λñ ⇒
  case ñ of r_ℤ n̂ ⇒ prom_𝒜 n̂
  | prom_ℤ n ⇒ prom_𝒜 (s_ℤ n)
  | if̂_ℤ b̂ ñ_1 ñ_2 ⇒ if̂_𝒜 b̂ (s̃_ℤ ñ_1) (s̃_ℤ ñ_2)

fn lîst : ℤ → ℕ = λk ⇒
  if k ≡ 0 then 0 else 1 + max 0 (1 + lîst (k−1))

fn tape͡ : 𝒜̃ → 𝒜 = λã ⇒
  case ã of prom_𝒜 â ⇒ â
  | if̂_𝒜 b̂ ã_1 ã_2 ⇒ mux b̂ (tape͡ ã_1) (tape͡ ã_2)
```

Fig. 7. Selected leaky types and functions in OIL

Translating oblivious types and operations. We represent every oblivious type as a single uniform type, the oblivious array $\mathcal{A}$. This array is essentially a secure "buffer" holding the private data. For example, $\widehat{\texttt{list}}$ k in the type signature of $s_{\texttt{list}}$, in Figure 5, is translated to this array type $\mathcal{A}$, regardless of the public view k. Even though oblivious types are all flat arrays in OIL, the rich typing information is not lost: we can still extract the needed private information by (securely) accessing the array using the sizes of oblivious types, such as the aforementioned $\widehat{\texttt{list}}$. Oblivious operations are translated into corresponding oblivious array operations. For example, an oblivious pair of private data is simply the concatenation of the two corresponding arrays, and destructing an oblivious pair amounts to taking a slice of the array using the two components' sizes. Section 4.2 describes the translation of other oblivious constructs, including injections into oblivious sums.

Translating tape semantics. Implementing the tape semantics is the main challenge in translating from TAYPE to OIL. Recall the three key ideas of the tape semantics. First, leaky operations, such as $\widehat{\texttt{if}}$, are themselves irreducible. Second, the surrounding context of $\widehat{\texttt{if}}$ is distributed into both branches. Third, the tape operation repairs potential leaks, by turning $\widehat{\texttt{if}}$ into mux.

To implement the first idea, we translate leaky types, e.g., $\mathbb{Z}_{\top}$, into a *leaky representation*, e.g., $\widetilde{\mathbb{Z}}$, a data type that explicitly represents expressions that may contain potential leaks. By convention, we use $\tilde{\cdot}$ as a visual cue for a leaky representation, its associated functions and variables. The leaky representations of oblivious array ($\widetilde{\mathcal{A}}$) and integers ($\widetilde{\mathbb{Z}}$) are shown in Figure 7. The only way to build a leaky oblivious data type is to promote a safe one or using a leaky conditional, so we simply encode these leaky operations as the constructors of its leaky representation $\widetilde{\mathcal{A}}$. $\widetilde{\mathbb{Z}}$ also includes both of these constructors, as well as its own retraction operation. This encoding trivially makes the leaky operations irreducible. Leaky representations of ADTs are built using a similar strategy (Section 4.2). Every leaky representation needs to have reified versions of prom and $\widehat{\texttt{if}}$, because $\uparrow$ and $\widehat{\texttt{if}}$ can be applied to any TAYPE type. During translation, they are instantiated using a process similar to typeclass resolution: the promotion of False in $\texttt{elem}_c$, for example, is resolved to $\texttt{prom}_{\mathbb{B}}$.

To distribute surrounding contexts into leaky constructs, we instrument the possible surrounding contexts to handle the leaky operations, by translating them into recursive functions following the tape semantics. For example, $s_{\mathbb{Z}}$ is translated to $\tilde{s}_{\mathbb{Z}}$, also shown in Figure 7. Observe the last case of $\tilde{s}_{\mathbb{Z}}$, which has recursive calls to itself in both $\widehat{\texttt{if}}_{\mathbb{Z}}$ branches: this aligns with our intuition from the execution trace of example (1). On the other hand, $\tilde{s}_{\mathbb{Z}}$'s handling of $r_{\mathbb{Z}}$ matches example (2).

Our solution to patching leaky computation without tape is encapsulated in the definition of $\widetilde{\texttt{tape}}$, shown in Figure 7. The function simply converts all reified $\widehat{\texttt{if}}$s into muxs, and is essentially a transcription of tape's evaluation rule.

To see how all these fit together, the initial expression in (1) is translated into the following OIL program: $\widetilde{\texttt{tape}}$ ($\tilde{s}_{\mathbb{Z}}$ ($\widehat{\texttt{if}}_{\mathbb{Z}}$ [True] ($\texttt{prom}_{\mathbb{Z}}$ 3) ($\texttt{prom}_{\mathbb{Z}}$ 4))). Readers can verify that evaluating this program using a standard semantics produces the same behavior seen in the execution trace in (1).

3 TAYPE, FORMALLY

This section describes the fully annotated core TAYPE language. This language is inspired by the core calculus $\lambda_{\text{OADT}\text{✚}}$ of Ye and Delaware [2022], but adds several features to aid its translation to

```
e,τ ::=                                                                                      Expressions
   | 𝔹 | ℤ | τ × τ                   standard types            | inl̂<τ> e | inr̂<τ> e              oblivious sum injection
   | 𝟙 | 𝔹̂ | ℤ̂ | τ ×̂ τ | τ +̂ τ      oblivious types           | case_τ e of (x, x) ⇒ e           product elim.
   | Πx:_l τ, τ | λx:_l τ ⇒ e        dependent function        | casê e:τ×̂τ of ⟨x, x⟩ ⇒ e         oblivious product elim.
   | () | b | n | x | C              literals, var. & constr.  | case_τ e of C x̄ ⇒ e              ADT case analysis
   | e ⊕ e | e ⊕̂ e                   (oblivious) binary op.    | s_𝔹 e | s_ℤ e                    primitive sections
   | e e                             application               | r_ℤ e                            primitive retraction
   | let x:_l τ = e in e             let binding               | if̂ e then e else e              leaky conditional
   | if_τ e then e else e            conditional               | casê_τ e:τ+̂τ of x⇒e | x⇒e       leaky oblivious sum elim.
   | mux e e e                       atomic conditional        | ↑e                               promotion
   | (e, e) | ⟨e, e⟩                 (oblivious) pair          | tape e                            tape operation

D ::=                                Global Definitions        v̂ ::= () | [b] | [n] | ⟨v̂, v̂⟩            Oblivious Values
   | data X = C̄ τ̄                    algebraic data type           | [inl<ω̂> v̂] | [inr<ω̂> v̂]
   | fn x:_l τ = e                   (recursive) function      v ::= if̂ [b] then v else v          Weak Values
   | obliv X̂ (x:τ) = τ               (recursive) oblivious type    | ↑v | r_ℤ v
l ::= ⊤ | ⊥                          Leakage Label                 | v̂ | b | n | (v, v)           Values
ω̂ ::= 𝟙 | 𝔹̂ | ℤ̂ | ω̂ ×̂ ω̂ | ω̂ +̂ ω̂      Oblivious Type Values         | λx:_l τ ⇒ e | C v̄
```

Fig. 8. Core Taype syntax: the annotations marked in gray are either omitted (e.g., promotion, labels) or optional (e.g., argument types to dependent functions) in the user-facing surface language; the expressions marked in brown are restricted to be variables in administrative normal form.

Oil, including oblivious products, label promotion, ML-style ADT definitions, and explicit and uniform label checking. The user-facing version of Taype allows for many annotations to be omitted; these annotations are automatically inferred by our bidirectional type checker (Section 3.5) before translation to Oil (Section 4).

3.1 Syntax

Figure 8 shows the syntax of core Taype. Types and terms are in the same syntactic class, as is common in dependently typed languages. By convention, we use e for terms and τ for types whenever possible. A core Taype program consists of a global context of ADTs, functions and oblivious types, defined using `data`, `fn` and `obliv` respectively. We use lower case x for function and variable names, C for constructors, upper case X for ADT names and $\widehat{X}$ for OADT names.

Taype features a number of oblivious types and constructs, including oblivious integers, booleans, sums, and conditionals. The primitive section functions $s_{\mathbb{B}}$ and $s_{\mathbb{Z}}$ "encrypt" boolean and integer values respectively. Oblivious products ($\widehat{\times}$) are built using ⟨·, ·⟩ and require both of their components to be oblivious and non-leaky. The atomic conditional `mux`, discussed in Section 2, fully evaluates both of its branches before taking an atomic step to its final result.

In core Taype, the arguments of dependent function types and lambda abstractions are annotated with a *leakage label* that indicates if they accept leaky inputs. We say that an Taype expression is *leaky* (i.e. has the label ⊤) if it contain potential leaks, e.g., uses some leaky operations, and say that it is *safe* otherwise. Standard conditional, product and ADT case analysis expressions are annotated with the result type, while the elimination forms for oblivious products and sums are also annotated with the type of the discriminee, to help with their translation. All of these annotations are inferred by our type checker. For brevity, we omit them from now if they can be inferred from the context.

Leaky conditionals, $\widehat{\texttt{case}}$ expressions, and `tape` operations play a key role in the semantics of Taype. The leaky conditional $\widehat{\texttt{if}}$ is similar to `mux`, but it allows its branches to be non-oblivious; $\widehat{\texttt{case}}$ analysis for oblivious sums is similar. The promotion operation ↑ explicitly converts a safe expression to a leaky one. Integer retraction $r_{\mathbb{Z}}$ would reveal its oblivious argument if implemented naively (as would the other leaky operations), but this is disallowed by the semantics of Taype.

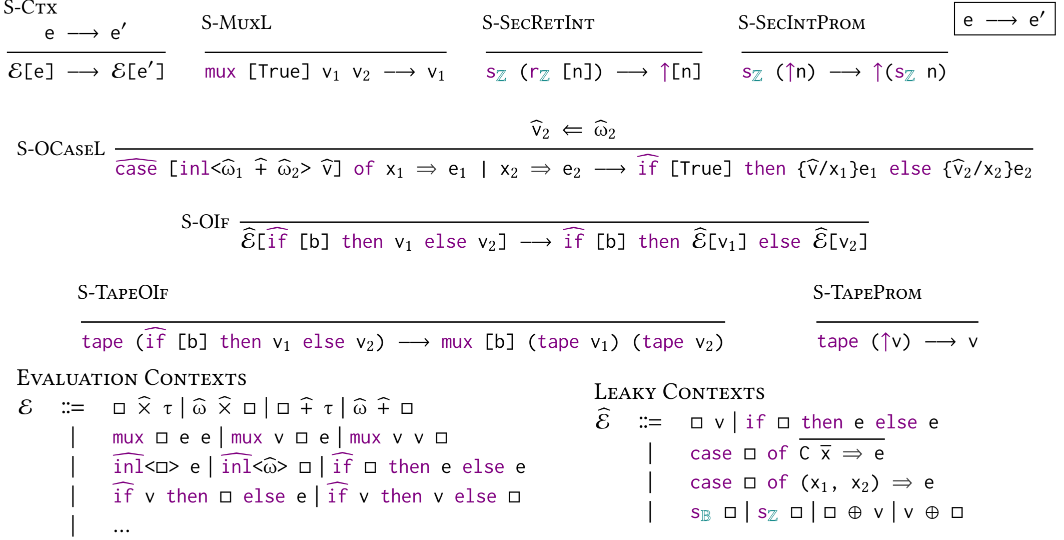


Fig. 9. Selected small-step semantics rules of core TAYPE

Oblivious types in TAYPE evaluate to *oblivious type values* ($\widehat{\omega}$), which provide a representation of private data that only depends on the public data. Oblivious terms evaluate to *oblivious values* ($\widehat{v}$); all values with the same oblivious type value are indistinguishable under our threat model. *Boxed values* like `[b]` represent these sorts of "encrypted" values, and can only appear at runtime. Normalized terms are defined as *weak values* (v) which may contain leaky operations. Standard values do not have any leaky operations and are thus always labelled as safe.

3.2 Semantics

Figure 9 shows a selection of the small-step operational semantics rules of core TAYPE.[3] The judgment $e \longrightarrow e'$ means e steps to e' under a fixed global context of definitions, which we elide. S-CTX takes a step in a subexpression according to an evaluation context $\mathcal{E}$, also given in Figure 9. Oblivious types are subject to reduction, as seen in the evaluation contexts involving $\widehat{\times}$, $\widehat{+}$ and $\widehat{\texttt{inl}}$. To prevent information leaks, all subexpressions of a `mux` are fully evaluated by first applying the S-CTX rule with the corresponding evaluation contexts, before `mux` itself can be reduced by either the S-MUXL or S-MUXR rule. The semantics of $\widehat{\texttt{if}}$ is similar. Note that an $\widehat{\texttt{if}}$ expression is in normal form once all its components are normalized, in order to avoid revealing its private condition.

The evaluation rules involving `tape` are one of the distinguishing features of TAYPE. S-OIF captures the idea that the leaky conditional $\widehat{\texttt{if}}$, while in normal form, can still make progress by distributing its context into both branches. *Leaky contexts* $\widehat{\mathcal{E}}$ define what contexts can be distributed in this manner: other contexts are either ruled out by the type system or not useful. S-TAPEOIF and S-TAPEPROM show how the `tape` operation repairs an expression with potential leaks. In addition to turning $\widehat{\texttt{if}}$ into `mux`, the enclosing `tape` is pushed inside the branches of `mux` in order to ensure any leaks they contain are also patched. On the other hand, S-TAPEPROM simply extracts the safe oblivious value from a promotion. In contrast to $\lambda_{\text{OADT}✚}$, TAYPE also includes new rules for promoted expressions. S-SECRETINT repairs the leaky operation $r_{\mathbb{Z}}$ by canceling it with $s_{\mathbb{Z}}$, for example, but it also promotes the resulting oblivious integer in order to preserve the leakage label. S-SECINTPROM shows how promotion interacts with $s_{\mathbb{Z}}$.

[3]The full set of reduction, typing, and kinding rules for core TAYPE are included in the appendix.

$$\text{T-Conv}\ \frac{\Gamma \vdash e :_l \tau' \quad \tau' \equiv \tau \quad \Gamma \vdash \tau :: *}{\Gamma \vdash e :_l \tau} \qquad \text{T-Abs}\ \frac{x{:}_l\tau, \Gamma \vdash e :_{l'} \tau' \quad \Gamma \vdash \tau :: *}{\Gamma \vdash \lambda x{:}_l\tau \Rightarrow e :_{l'} \Pi x{:}_l\tau,\ \tau'}$$

$$\text{T-App}\ \frac{\Gamma \vdash e_1 :_{l_1} \Pi x{:}_{l_2}\tau_2,\ \tau_1 \quad \Gamma \vdash e_2 :_{l_2} \tau_2}{\Gamma \vdash e_1\ e_2 :_{l_1} \{e_2/x\}\tau_1} \qquad \boxed{\Gamma \vdash e :_l \tau}$$

$$\text{T-Pair}\ \frac{\Gamma \vdash e_1 :_l \tau_1 \quad \Gamma \vdash e_2 :_l \tau_2}{\Gamma \vdash (e_1,\ e_2) :_l \tau_1 \times \tau_2} \qquad \text{T-PCaseNoDep}\ \frac{\Gamma \vdash e_0 :_{l_0} \tau_1 \times \tau_2 \quad l_0 \sqsubseteq l \quad x_1{:}_{l_0}\tau_1, x_2{:}_{l_0}\tau_2, \Gamma \vdash e :_l \tau}{\Gamma \vdash \mathtt{case}_\tau\ e_0\ \mathtt{of}\ (x_1,\ x_2) \Rightarrow e :_l \tau}$$

$$\text{T-IfNoDep}\ \frac{\Gamma \vdash e_0 :_{l_0} \mathbb{B} \quad l_0 \sqsubseteq l \quad \Gamma \vdash e_1 :_l \tau \quad \Gamma \vdash e_2 :_l \tau}{\Gamma \vdash \mathtt{if}_\tau\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 :_l \tau}$$

$$\text{T-Mux}\ \frac{\Gamma \vdash e_0 :_\bot \widehat{\mathbb{B}} \quad \Gamma \vdash \tau :: *^{O} \quad \Gamma \vdash e_1 :_\bot \tau \quad \Gamma \vdash e_2 :_\bot \tau}{\Gamma \vdash \mathtt{mux}\ e_0\ e_1\ e_2 :_\bot \tau} \qquad \text{T-OPair}\ \frac{\Gamma \vdash e_1 :_\bot \tau_1 \quad \Gamma \vdash e_2 :_\bot \tau_2 \quad \Gamma \vdash \tau_1 :: *^{O} \quad \Gamma \vdash \tau_2 :: *^{O}}{\Gamma \vdash \langle e_1,\ e_2\rangle :_\bot \tau_1 \mathbin{\widehat{\times}} \tau_2}$$

$$\text{T-SecInt}\ \frac{\Gamma \vdash e :_l \mathbb{Z}}{\Gamma \vdash s_{\mathbb{Z}}\ e :_l \widehat{\mathbb{Z}}} \qquad \text{T-RetInt}\ \frac{\Gamma \vdash e :_\bot \widehat{\mathbb{Z}}}{\Gamma \vdash r_{\mathbb{Z}}\ e :_\top \mathbb{Z}}$$

$$\text{T-OIf}\ \frac{\Gamma \vdash e_0 :_\bot \widehat{\mathbb{B}} \quad \Gamma \vdash e_1 :_\top \tau \quad \Gamma \vdash e_2 :_\top \tau}{\Gamma \vdash \widehat{\mathtt{if}}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 :_\top \tau} \qquad \text{T-Promote}\ \frac{\Gamma \vdash e :_\bot \tau}{\Gamma \vdash {\uparrow}e :_\top \tau} \qquad \text{T-Tape}\ \frac{\Gamma \vdash e :_\top \tau \quad \Gamma \vdash \tau :: *^{O}}{\Gamma \vdash \mathtt{tape}\ e :_\bot \tau}$$

Fig. 10. Selected core Taype typing rules

To see how these rules work, consider a core Taype version of example (1) from Section 2, which produces the following execution trace:

$$\begin{aligned}
&\mathtt{tape}\ (s_{\mathbb{Z}}\ (\widehat{\mathtt{if}}\ [\mathtt{True}]\ \mathtt{then}\ {\uparrow}3\ \mathtt{else}\ {\uparrow}4)) \longrightarrow \mathtt{tape}\ (\widehat{\mathtt{if}}\ [\mathtt{True}]\ \mathtt{then}\ s_{\mathbb{Z}}\ {\uparrow}3\ \mathtt{else}\ s_{\mathbb{Z}}\ {\uparrow}4)\\
&\longrightarrow \mathtt{tape}\ (\widehat{\mathtt{if}}\ [\mathtt{True}]\ \mathtt{then}\ {\uparrow}(s_{\mathbb{Z}}\ 3)\ \mathtt{else}\ s_{\mathbb{Z}}\ {\uparrow}4) \longrightarrow \mathtt{tape}\ (\widehat{\mathtt{if}}\ [\mathtt{True}]\ \mathtt{then}\ {\uparrow}[3]\ \mathtt{else}\ s_{\mathbb{Z}}\ {\uparrow}4)\\
&\longrightarrow^* \mathtt{tape}\ (\widehat{\mathtt{if}}\ [\mathtt{True}]\ \mathtt{then}\ {\uparrow}[3]\ \mathtt{else}\ {\uparrow}[4]) \rightarrow \mathtt{mux}\ [\mathtt{True}]\ (\mathtt{tape}\ {\uparrow}[3])\ (\mathtt{tape}\ {\uparrow}[4])\\
&\longrightarrow^* \mathtt{mux}\ [\mathtt{True}]\ [3]\ [4] \longrightarrow [3]
\end{aligned}$$

The S-OCaseL rule reduces a leaky case analysis of an oblivious sum to an $\widehat{\mathtt{if}}$ using the discriminee's private tag. The pattern variable x in the "correct" branch is of course instantiated with the injection payload, while the one in the "wrong" branch is instantiated with an arbitrary oblivious value of the right type. This arbitrary value is synthesized by the judgment $\widehat{v} \Leftarrow \widehat{\omega}$ (the details of this relation are included in the appendix). For example, if the discriminee of a leaky case is $[\mathtt{inl}{<}\widehat{\mathbb{Z}} \mathbin{\widehat{+}} \widehat{\mathbb{Z}} \mathbin{\widehat{\times}} \widehat{\mathbb{Z}}{>}\ 1]$, the pattern variable in the second branch can be substituted by $\langle[0],\ [0]\rangle$, $\langle[0],\ [1]\rangle$, or any other oblivious pair of oblivious integers.

3.3 Type System

Figure 10 shows an illustrative subset of the typing rules of core Taype. The judgment $\Gamma \vdash e :_l \tau$ types the expression e with type τ and leakage label l, under the typing context Γ (and an elided global typing context). Some typing rules refer to the kinding judgment $\Gamma \vdash \tau :: \kappa$, which also classifies the security of a type; oblivious types have the kind $*^{O}$, for example.

Taype features a security-type system [Sabelfeld and Myers 2003] that ensures well-typed programs protect their private data. To do so, this type system enforces a few key policies. First, oblivious types can only be built from oblivious types, which is enforced by the kinding rules. Otherwise, an attacker can infer the private tag of an oblivious sum, such as $\mathbb{B} \mathbin{\widehat{+}} \mathbb{Z}$, by observing the payload. Oblivious products have the same requirement, although this is mainly to aid in translation. Second, oblivious control flow constructs like mux can only be applied to oblivious terms, otherwise their public result could reveal information about their condition. As an example, $\mathtt{mux}\ [b]\ 1\ 2$ is ill-typed, because an attacker can learn the value of b by observing its result. This policy is enforced by the kinding assumptions of the form $\Gamma \vdash \tau :: *^{O}$ in T-Mux and T-OPair. Third, types are not allowed to depend on leaky terms. The type $\mathtt{if}\ r_{\mathbb{Z}}\ [0] \equiv 0\ \mathtt{then}\ \mathbb{1}\ \mathtt{else}\ \widehat{\mathbb{B}}$ is not valid,

for example, since the leaks in the condition can not be repaired. Thus, we require that any terms appearing in types to be labeled as non-leaky ($\bot$). Fourth, the argument to tape must be oblivious (T-TAPE). This ensures that leaky terms will *eventually* reduce to an oblivious value or a $\widehat{\text{if}}$ tree of oblivious values that can then be repaired by, e.g., S-TAPEPROM or S-TAPEOIF. Intuitively, the $\bot$ label in the conclusion of T-TAPE signifies that the taped expression can be treated as non-leaky by its surrounding computation, as all leaks have been "patched up". Finally, all oblivious components in the typing rules have the $\bot$ label. All the labels in T-MUX and T-OPAIR are $\bot$, and the oblivious condition of $\widehat{\text{if}}$ (T-OIF) is also safe, for example. While this requirement is not crucial for security, it simplifies the type system and aids in our translation to OIL. Note that we can always apply tape to leaky oblivious expressions to make them safe, so this design does not harm the expressivity of well-typed TAYPE programs.

In addition to the above policies, TAYPE's type system imposes three more requirements that help our translation. First, safe terms must be explicitly converted to leaky ones using $\uparrow$. Thus, T-CONV requires convertible expressions to have the same label. Second, we usually require subexpressions to have the same label: the two components in T-PAIR have the same label l in TAYPE, for example. T-IFNODEP similarly requires both branches to have the same label. Its condition, however, is permitted to have a lower label. A similar requirement is particularly important for the case analysis of products and ADTs: each branch needs to use its pattern variables in a manner that is at least as safe as the discriminee. Third, we require all *possibly* leaky subexpressions to be labelled as leaky. The branches in T-OIF and the argument to T-TAPE have label $\top$, even though they can technically also be typed at $\bot$: applying these rules to an expression with a safe subterm requires explicit promotion. Note that programmers do not need to do these explicit label conversion in the surface language, as $\uparrow$ is automatically inserted by the typing algorithm presented in Section 3.5.

3.4 Type Soundness and Obliviousness

Given a well-typed global context, core TAYPE enjoys standard progress and preservation properties. Its type system also provides a strong security guarantee: an adversary cannot infer any private information from a well-typed core TAYPE program, even when they can observe each of its execution steps.

THEOREM 3.1 (OBLIVIOUSNESS). *If* $e_1 \approx e_2$ *and* $\cdot \vdash e_1 :_{l_1} \tau_1$ *and* $\cdot \vdash e_2 :_{l_2} \tau_2$, *then*
(1) $e_1 \longrightarrow^n e_1'$ *if and only if* $e_2 \longrightarrow^n e_2'$ *for some* e_2'.
(2) if $e_1 \longrightarrow^n e_1'$ *and* $e_2 \longrightarrow^n e_2'$, *then* $e_1' \approx e_2'$.

Here, $e_1 \approx e_2$ means the two expressions are *indistinguishable*, i.e. they only differ in their oblivious values, and $e \longrightarrow^n e'$ means e reduces to e' in exactly n steps. Intuitively, the obliviousness theorem says that a pair of well-typed core TAYPE programs that are indistinguishable produce traces that are pairwise indistinguishable.

We have formalized a version of core TAYPE in Coq, including proofs of soundness and obliviousness for the calculus, based on the mechanization from Ye and Delaware [2022]. In contrast to that development, this calculus includes the new features of TAYPE: oblivious products, label promotion and explicit and uniform label checking.

3.5 Surface Language and Bidirectional Type Checker

The source language of our compiler is a more user friendly version of core TAYPE. This language allows type annotations to be omitted, and does not require label annotations or explicit promotion operations. Its syntax is effectively that of Figure 8 with the gray annotations removed. Our type checker elaborates programs in this surface language into fully annotated core TAYPE programs in ANF [Flanagan et al. 1993]. Our inference algorithm is not sophisticated: unlike other dependent

type systems, it does not support unification, for example. Nevertheless, it is capable of checking all the case studies and benchmarks in our experiments (Section 6) without any type or label annotations, except for top-level definitions.

Due to space concerns, the full typing algorithm is elided.[4] At a high level, like standard bidirectional type checkers, our type checker operates in an *inference mode* and a *checking mode*. In inference mode, the algorithm infers the type of an expression (bottom-up), while in checking mode, the algorithm checks the expression against an expected type by propagating information to subexpressions as deeply as possible (top-down). Our type checker always starts with checking mode, as all top-level definitions are annotated with their type.

The main challenge to algorithmic type checking are dependent conditionals and ADT case analysis, specifically inferring their *implicit motives*. Since dependent types in Taype are oblivious types, they are more restricted than the ones in most dependent type systems. To see how, consider the expression: `if x then (λb:`$\widehat{\mathbb{B}}$ `⇒` $\widehat{\texttt{if}}$ `b then 1 else 0) else (λn:`$\widehat{\mathbb{Z}}$ `⇒` $r_{\mathbb{Z}}$ `n)`. Ignoring labels, the left branch of this conditional has type $\widehat{\mathbb{B}} \to \mathbb{Z}$, while the right one has type $\widehat{\mathbb{Z}} \to \mathbb{Z}$. In many dependent type systems, this expression can simply be typed with `if x then` $\widehat{\mathbb{B}} \to \mathbb{Z}$ `else` $\widehat{\mathbb{Z}} \to \mathbb{Z}$. However, this type is not well-kinded in Taype! The type-level computation, `if` in this case, is only defined over oblivious types, which the types in the branches are clearly not. The correct type of this expression has to be `(if x then` $\widehat{\mathbb{B}}$ `else` $\widehat{\mathbb{Z}}$`)` $\to \mathbb{Z}$. The same problem also occurs in dependent ADT case analysis, and when the branches have product types. Our type checker is equipped with special inference and checking rules for handling these cases.

Our bidirectional type checker also infers leakage labels. In contrast to type annotations, label annotations are not required even for top-level function signatures:[5] they are instead derived from a function *attribute*, which indicates the purpose of a function. A function can be marked as either *section*, *retraction*, or *safe* using the `#[attribute]` syntax, as shown in Figure 5. A function without an attribute, such as `elem`, implements the program logic in the conventional fragment of Taype. Such functions and their arguments are always labelled as leaky, since they have to accept retracted values to work with the recipe from Section 2.1. Any intermediate labels in the bodies of such functions can also be reliably inferred to be leaky, as these functions do not mention oblivious types or public views directly. As a result, a programmer can write the functionality as in a conventional functional language. On the other hand, a function annotated with `#[section]`, e.g., $s_{\texttt{list}}$, defines a section function. Its public view argument obviously has a safe label, while its data argument (e.g., `list`) has a leaky label. A section function itself is safe, signaling that all potential leaks have been patched. Conversely, a function, annotated with `#[retraction]`, labels its arguments as safe, but itself has leaky label. Lastly, functions annotated with `#[safe]`, e.g., $\widehat{\texttt{elem}}$, are secure functions. These constitute the API of a secure library, so their arguments and the functions themselves are assigned safe labels. Note that while the labels in a function's signature are determined by its attribute, any intermediate labels in its body need to be inferred. Similar to types, labels are inferred bidirectionally. When an inferred label is checked against a label, the checker will insert a promotion if the expected label is more restrictive than the inferred label, and reject the program when the expected label is less restrictive than the inferred label.

4 OIL AND TRANSLATION

This section describes the OADT intermediate language, Oil, and its translation from Taype. The main challenge is how to encode the features of Taype that Oil lacks, including dependent types, leaky operations ($\widehat{\texttt{if}}$ and `tape`), and, most importantly, its tape semantics.

[4] A representative selection of the bidirectional type checking rules are available in the appendix.

[5] In fact, our surface syntax does not allow users to provide label annotations.

e ::=	Expressions
| () | b | n | x | C	literals, variable and constructor
| e ⊕ e | e $\widehat{\oplus}$ e	(oblivious) integer operations
| λx ⇒ e | e e	function abs. and app.
| let x = e in e	let binding
| if e then e else e	conditional
| mux e e e	atomic conditional
| (e, e)	pair
| case e of (x, x) ⇒ e	product elimination
| case e of $\overline{\texttt{C x} \Rightarrow \texttt{e}}$	ADT case analysis
| $s_{\mathbb{B}}$ e | $s_{\mathbb{Z}}$ e	primitive sections
| $\mathcal{A}$(e) | e ++ e | e(e, e)	oblivious array operations
| ...	size ($\mathbb{N}$) operations, omitted

τ ::=	Types
| $\mathbb{1}$ | $\mathbb{B}$ | $\mathbb{Z}$	base types
| $\mathcal{A}$	oblivious array
| $\mathbb{N}$	size type
| α	type variable
| X	ADT variable
| τ × τ	product type
| τ → τ	function type

D ::=	Global Definitions
| data X[$\overline{\alpha}$] = $\overline{\texttt{C}\ \overline{\tau}}$	algebraic data type
| fn x[$\overline{\alpha}$]:τ = e	(recursive) function

Fig. 11. Oil source syntax

$$\llbracket \mathbb{Z} \rrbracket_\bot = \mathbb{Z} \qquad \llbracket \mathtt{X} \rrbracket_\bot = \mathtt{X} \qquad \llbracket \tau_1 \times \tau_2 \rrbracket_\bot = \llbracket \tau_1 \rrbracket_\bot \times \llbracket \tau_2 \rrbracket_\bot \qquad \llbracket \Pi \mathtt{x}{:}_l \tau_1,\ \tau_2 \rrbracket_\bot = \llbracket \tau_1 \rrbracket_l \to \llbracket \tau_2 \rrbracket_\bot$$

$$\llbracket \mathbb{1} \rrbracket_\bot = \llbracket \widehat{\mathbb{Z}} \rrbracket_\bot = \llbracket \tau_1 \mathbin{\widehat{\times}} \tau_2 \rrbracket_\bot = \llbracket \tau_1 \mathbin{\widehat{+}} \tau_2 \rrbracket_\bot = \llbracket \widehat{\mathtt{X}}\ \mathtt{e} \rrbracket_\bot = \llbracket \mathtt{if}\ \ldots \rrbracket_\bot = \llbracket \mathtt{let}\ \ldots \rrbracket_\bot = \llbracket \mathtt{case}\ \ldots \rrbracket_\bot = \mathcal{A} \qquad \boxed{\llbracket \tau \rrbracket_\bot}$$

$$\llbracket \mathbb{Z} \rrbracket_\top = \widetilde{\mathbb{Z}} \qquad \llbracket \mathtt{X} \rrbracket_\top = \widetilde{\mathtt{X}} \qquad \llbracket \tau_1 \times \tau_2 \rrbracket_\top = \llbracket \tau_1 \rrbracket_\top \mathbin{\widetilde{\times}} \llbracket \tau_2 \rrbracket_\top \qquad \llbracket \Pi \mathtt{x}{:}_l \tau_1,\ \tau_2 \rrbracket_\top = \llbracket \tau_1 \rrbracket_l \to \llbracket \tau_2 \rrbracket_\top$$

$$\llbracket \mathbb{1} \rrbracket_\top = \llbracket \widehat{\mathbb{Z}} \rrbracket_\top = \llbracket \tau_1 \mathbin{\widehat{\times}} \tau_2 \rrbracket_\top = \llbracket \tau_1 \mathbin{\widehat{+}} \tau_2 \rrbracket_\top = \llbracket \widehat{\mathtt{X}}\ \mathtt{e} \rrbracket_\top = \llbracket \mathtt{if}\ \ldots \rrbracket_\top = \llbracket \mathtt{let}\ \ldots \rrbracket_\top = \llbracket \mathtt{case}\ \ldots \rrbracket_\top = \widetilde{\mathcal{A}} \qquad \boxed{\llbracket \tau \rrbracket_\top}$$

Fig. 12. Selected rules for translating core Taype types to Oil types

4.1 Syntax, Semantics and Type System

Figure 11 shows the syntax of Oil. It is mostly a standard ML-style language with rank-1 polymorphism, extended with an *oblivious array* type and its operations. An oblivious array $\mathcal{A}$ is essentially a "buffer" holding all the private data in a joint computation. The elements of this array are the oblivious representation (usually encrypted values) of members of some fixed finite field. To remain agnostic to the underlying cryptographic protocol, Oil does not place any restrictions on the oblivious representation or the finite field, so the array can hold the encryption of bits, or shared secrets of 64-bit integers, for example. Conceptually, each array element is simply an oblivious integer that encodes a piece of the private data, such as an oblivious integer, the tag of an oblivious injection, or an oblivious boolean. Programs create an array of size n using $\mathcal{A}$(n), concatenate two arrays using ++, and take a slice of n elements starting at offset m in array a via a(m, n).

Like Taype, Oil includes oblivious operations, but these operations are restricted to take and produce oblivious arrays, as this is the *only* oblivious type in Oil. The section operations for base types, $s_{\mathbb{B}}$ and $s_{\mathbb{Z}}$, for example, return a singleton array containing the "encrypted" result.

Types and global definitions are also standard, but Oil also includes the *size type*, $\mathbb{N}$, for array offsets and lengths. Oil has a standard CBV semantics and type system. The semantics of array operations that use out-of-bound indices (e.g., slicing) is undefined: this should never happen if translated from a well-typed Taype program.

4.2 Translating from Taype to Oil

Our translation from Taype to Oil is syntax- and type-directed, and uses the leakage label to identify and repair potential leaks in the program. The translation assumes the source program is in *administrative normal form* (ANF), restricting the brown-colored expressions e in Figure 8 to be variables. The algorithm roughly consists of three components: translating Taype types to Oil types (Figure 12), translating Taype expressions to Oil expressions (Figure 13), and translating Taype oblivious types to Oil expressions of the size type (Figure 16).

Translating Types. Figure 12 shows the translation of a Taype type τ to an Oil type, guided by a leakage label l. With the ⊥ label, public types are translated as they are or congruently, as expected.

$$\textsc{TR-SecInt}\ \frac{}{\Gamma \vdash \mathtt{s}_{\mathbb{Z}}\ \mathtt{x} \rightsquigarrow_l \begin{cases} \mathtt{s}_{\mathbb{Z}}\ \mathtt{x} & \text{if } l = \bot \\ \widetilde{\mathtt{s}_{\mathbb{Z}}}\ \mathtt{x} & \text{if } l = \top \end{cases}} \qquad \textsc{TR-Pair}\ \frac{}{\Gamma \vdash (\mathtt{x}_1,\ \mathtt{x}_2) \rightsquigarrow_l \begin{cases} (\mathtt{x}_1,\ \mathtt{x}_2) & \text{if } l = \bot \\ \widetilde{\mathtt{pair}}\ \mathtt{x}_1\ \mathtt{x}_2 & \text{if } l = \top \end{cases}} \qquad \textsc{TR-App}\ \frac{}{\Gamma \vdash \mathtt{x}_2\ \mathtt{x}_1 \rightsquigarrow_l \mathtt{x}_2\ \mathtt{x}_1} \qquad \boxed{\Gamma \vdash \mathtt{e} \rightsquigarrow_l \dot{\mathtt{e}}}$$

$$\textsc{TR-Abs}\ \frac{\mathtt{x}:_{l_1}\tau_1, \Gamma \vdash \mathtt{e} \rightsquigarrow_l \dot{\mathtt{e}}}{\Gamma \vdash \lambda \mathtt{x}:_{l_1}\tau_1 \Rightarrow \mathtt{e} \rightsquigarrow_l \lambda \mathtt{x} \Rightarrow \dot{\mathtt{e}}} \qquad \textsc{TR-If}\ \frac{\mathtt{x}_0 :_{l_0} \mathbb{B} \in \Gamma \qquad \Gamma \vdash \mathtt{e}_1 \rightsquigarrow_l \dot{\mathtt{e}}_1 \qquad \Gamma \vdash \mathtt{e}_2 \rightsquigarrow_l \dot{\mathtt{e}}_2}{\Gamma \vdash \mathtt{if}_\tau\ \mathtt{x}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 \rightsquigarrow_l \begin{cases} \mathtt{if}\ \mathtt{x}_0\ \mathtt{then}\ \dot{\mathtt{e}}_1\ \mathtt{else}\ \dot{\mathtt{e}}_2 & \text{if } l_0 = \bot \\ \widetilde{\mathtt{if}}\ \widehat{\mathtt{if}}(\tau)\ \mathtt{x}_0\ \dot{\mathtt{e}}_1\ \dot{\mathtt{e}}_2 & \text{if } l_0 = \top \end{cases}}$$

(a) Translating standard constructs

$$\textsc{TR-Unit}\ \frac{}{\Gamma \vdash () \rightsquigarrow_\bot \mathcal{A}(0)} \qquad \textsc{TR-OPair}\ \frac{}{\Gamma \vdash \langle \mathtt{x}_1,\ \mathtt{x}_2 \rangle \rightsquigarrow_\bot \mathtt{x}_1 +\!\!+\ \mathtt{x}_2} \qquad \textsc{TR-OInl}\ \frac{\Gamma \vdash \tau_1 \rightsquigarrow \mathtt{s}_1 \qquad \Gamma \vdash \tau_2 \rightsquigarrow \mathtt{s}_2}{\Gamma \vdash \widehat{\mathtt{inl}}\texttt{<}\tau_1\ \widehat{+}\ \tau_2\texttt{>}\ \mathtt{x} \rightsquigarrow_\bot \widehat{\mathtt{inl}}\ \mathtt{s}_1\ \mathtt{s}_2\ \mathtt{x}} \qquad \textsc{TR-RetInt}\ \frac{}{\Gamma \vdash \mathtt{r}_{\mathbb{Z}}\ \mathtt{x} \rightsquigarrow_\top \mathtt{r}_{\mathbb{Z}}\ \mathtt{x}}$$

$$\textsc{TR-Tape}\ \frac{}{\Gamma \vdash \mathtt{tape}\ \mathtt{x} \rightsquigarrow_\bot \widetilde{\mathtt{tape}}\ \mathtt{x}} \qquad \textsc{TR-Promote}\ \frac{\mathtt{x}:_\bot\tau \in \Gamma}{\Gamma \vdash \uparrow\!\mathtt{x} \rightsquigarrow_\top \mathtt{prom}(\tau)\ \mathtt{x}} \qquad \textsc{TR-OIf}\ \frac{\mathtt{x}_1:_\top\tau \in \Gamma}{\Gamma \vdash \widehat{\mathtt{if}}\ \mathtt{x}_0\ \mathtt{then}\ \mathtt{x}_1\ \mathtt{else}\ \mathtt{x}_2 \rightsquigarrow_\top \widehat{\mathtt{if}}(\tau)\ \mathtt{x}_0\ \mathtt{x}_1\ \mathtt{x}_2}$$

$$\textsc{TR-OPCase}\ \frac{\mathtt{x}_1:_\bot\tau_1, \mathtt{x}_2:_\bot\tau_2, \Gamma \vdash \mathtt{e} \rightsquigarrow_l \dot{\mathtt{e}} \qquad \Gamma \vdash \tau_1 \rightsquigarrow \mathtt{s}_1 \qquad \Gamma \vdash \tau_2 \rightsquigarrow \mathtt{s}_2}{\Gamma \vdash \widehat{\mathtt{case}}\ \mathtt{x}_0:\tau_1 \widehat{\times} \tau_2\ \mathtt{of}\ \langle \mathtt{x}_1,\ \mathtt{x}_2 \rangle \Rightarrow \mathtt{e} \rightsquigarrow_l \mathtt{let}\ \mathtt{x}_1 = \mathtt{x}_0(0,\ \mathtt{s}_1)\ \mathtt{in}\ \mathtt{let}\ \mathtt{x}_2 = \mathtt{x}_0(\mathtt{s}_1,\ \mathtt{s}_2)\ \mathtt{in}\ \dot{\mathtt{e}}}$$

$$\textsc{TR-OCase}\ \frac{\mathtt{x}:_\bot\tau_1, \Gamma \vdash \mathtt{e}_1 \rightsquigarrow_\top \dot{\mathtt{e}}_1 \qquad \mathtt{x}:_\bot\tau_2, \Gamma \vdash \mathtt{e}_2 \rightsquigarrow_\top \dot{\mathtt{e}}_2 \qquad \Gamma \vdash \tau_1 \rightsquigarrow \mathtt{s}_1 \qquad \Gamma \vdash \tau_2 \rightsquigarrow \mathtt{s}_2}{\Gamma \vdash \widehat{\mathtt{case}}_\tau\ \mathtt{x}_0:\tau_1 \widehat{+} \tau_2\ \mathtt{of}\ \mathtt{x} \Rightarrow \mathtt{e}_1 \mid \mathtt{x} \Rightarrow \mathtt{e}_2 \rightsquigarrow_\top \begin{array}{l} \mathtt{let}\ \mathtt{tag} = \mathtt{x}_0(\mathtt{max}\ \mathtt{s}_1\ \mathtt{s}_2,\ 1)\ \mathtt{in} \\ \widehat{\mathtt{if}}(\tau)\ \mathtt{tag}\ (\mathtt{let}\ \mathtt{x} = \mathtt{x}_0(0,\ \mathtt{s}_1)\ \mathtt{in}\ \dot{\mathtt{e}}_1) \\ \qquad (\mathtt{let}\ \mathtt{x} = \mathtt{x}_0(0,\ \mathtt{s}_2)\ \mathtt{in}\ \dot{\mathtt{e}}_2) \end{array}}$$

(b) Translating leaky and oblivious constructs

Fig. 13. Selected rules for translating core Taype expressions to Oil expressions

Oblivious types, in contrast, are always converted to an oblivious array in Oil. The rich typing information of an oblivious type is not thrown away however: as we shall see, this information is used to implement oblivious array operations. Dependent function types are translated to their nondependent counterpart, with the label on the parameter type dictating its translation.

The translation of types under the $\top$ label is more involved. To understand why, recall that an expression with this label may contain a potentially leaky subexpression which should be repaired via `tape`. Thus, its Oil counterpart must be equipped with a similar mechanism capable of patching leaks. Our solution is to explicitly capture the insecure operations associated with a particular leaky type in its Oil representation, and to insert repairs for each kind of leak when translating a leaky expression. We call this first component a *leaky representation*. As an example, an integer expression can have three kinds of leaks: it could be a retraction of a secure integer $\mathtt{r}_{\mathbb{Z}}$, it could be a leaky conditional $\widehat{\mathtt{if}}$, or it could be the promotion of a plaintext integer $\uparrow$. The corresponding leaky representation, $\widetilde{\mathbb{Z}}$, is shown in Figure 7, and contains a constructor for each of these cases. As every leaky type can leak information via $\uparrow$ and $\widehat{\mathtt{if}}$, all leaky representations should be equipped with a reified form of these leaky expressions. Thus, every leaky representation (with its safe counterpart) forms a *leaky structure*, with operations `prom` and $\widetilde{\mathtt{if}}$ for $\uparrow$ and $\widehat{\mathtt{if}}$ respectively. From an implementation perspective, the leaky structure operations define a *typeclass*, so we call a particular `prom` and $\widetilde{\mathtt{if}}$ *instances* of this typeclass. As one example, the constructors of $\widetilde{\mathbb{Z}}$ trivially provide the necessary instances. As another example, Figure 7 also shows $\widetilde{\mathcal{A}}$, the leaky representation of an oblivious array; its leaky instance is similarly defined by the two constructors of this type.

```
data list = Nil | Cons ℤ list
data list~ = Nil~ | Cons~ ℤ~ list~
        | prom_list list | if^_list 𝒜 list~ list~
fn case~_list [γ~] : (𝒜 → γ~ → γ~ → γ~) → list~ →
                    γ~ → (ℤ~ → list~ → γ~) → γ~ =
  λif^_γ xs~ f₁ f₂ ⇒
    case xs~ of Nil~ ⇒ f₁
    | Cons~ x~ xs~' ⇒ f₂ x~ xs~'
    | prom_list xs ⇒
      case xs of Nil ⇒ f₁
      | Cons x xs' ⇒ f₂ (prom_ℤ x) (prom_list xs')
    | if^_list b^ xs~₁ xs~₂ ⇒
      if^_γ b^ (case~_list if^_γ xs~₁ f₁ f₂)
            (case~_list if^_γ xs~₂ f₁ f₂)
```

Fig. 14. Leaky structures for lists

In general, the $\widehat{\texttt{if}}$ instances are usually constructors, as a leaky conditional needs to be irreducible to avoid leaking its private condition. The promotion instances are also constructors in our translation, although in general they need not be.[6] Of course, our translation must also explain how to use leaky values, i.e. how to interpret the corresponding *elimination forms* of τ. To illustrate this, consider the leaky structure for list shown in Figure 14. The leaky representation of lists includes constructors for Cons and Nil, i.e. the introduction forms of list. Its leaky elimination form, $\widetilde{\texttt{case}}_{\texttt{list}}$, is straightforward: the $\texttt{prom}_{\texttt{list}}$ branch promotes the arguments of each constructor before applying the "alternative functions", and the $\widehat{\texttt{if}}_{\texttt{list}}$ branch essentially encodes the tape semantics rule S-OIf, specialized to the leaky context of case expressions, $\texttt{case} \ \square \ \texttt{of} \ \overline{\texttt{C} \ \overline{\texttt{x}} \to \texttt{e}}$.

A similar recipe is used to derive the leaky representation and its associated functions for other types: the introduction forms are encoded as constructors with the $\widehat{\texttt{if}}$ and prom instances, and the elimination forms capture the idea of distributing the corresponding leaky context into the $\widehat{\texttt{if}}$ branches and how $\uparrow$ interacts with this context. While the leaky structures of builtin and arrow types are defined in the Oil prelude, the ones for user-defined ADTs are generated using the algorithm in Figure 15. This is how the leaky definition of list in Figure 14 was generated, for example. An ADT's introduction forms are its constructors, so the leaky representation just renames them, with the constructor argument types translated with label $\top$. The $\widetilde{\texttt{case}}_{\texttt{X}}$ function encodes the elimination form of ADTs, using a list of functions corresponding to branches of a case expression. The $\texttt{prom}_{\texttt{X}}$ branch relies on the instance resolution procedure $\texttt{prom}(\cdot)$ to promote constructor arguments.

```
data X = C ⟦τ⟧⊥ (overlined)
data X~ = C~ ⟦τ⟧⊤ (overlined) | prom_X X | if^_X 𝒜 X~ X~
fn case~_X [γ~] : (𝒜 → γ~ → γ~ → γ~) → X~ →
                  (⟦τ⟧⊤ (overlined) → γ~) (overlined) → γ~ =
  λif^_γ x~ f̄ ⇒
    case x~ of C~ x̄~ ⇒ f x̄~ (overlined)
    | prom_X x ⇒ case x of C x̄ ⇒ f (prom(τ) x) (overlined)
    | if^_X b^ x~₁ x~₂ ⇒
      if^_γ b^ (case~_X if^_γ x~₁ f̄) (case~_X if^_γ x~₂ f̄)
```

Fig. 15. Generating leaky ADT definitions

Translating Expressions. We now present our translation from Taype to Oil expressions. As with our translation of types, the translation of expressions is given as a judgment $\Gamma \vdash \texttt{e} \rightsquigarrow_l \dot{\texttt{e}}$, that is indexed by a leakage label l which guides the translation. Figure 13a illustrates how l drives the translation of standard constructs: if l identifies an expression as safe, it is simply translated congruently. On the other hand, if an expression is marked as leaky, the translation relies on the leaky context of the expression to patch any leaks. This strategy can be seen in the TR-SecInt rule: using this rule to translate a leaky $\texttt{s}_{\mathbb{Z}} \ \texttt{e}$ expression delegates any repairs to $\widetilde{\texttt{s}}_{\mathbb{Z}}$. Translating lambda abstractions (TR-Abs) and applications (TR-App) is straightforward. TR-Pair shows why we require uniform labels in subexpressions: the components of a pair marked as leaky must also be leaky, as $\widetilde{\texttt{pair}}$ takes the leaky representations as arguments, similar to $\widetilde{\texttt{Cons}}$ from Figure 14. The translation of if (TR-If) differs from the other rules in that the label of its discriminee dictates

[6] Intuitively, these two instances are generated "for free", though not in the algebraic sense. The only free leaky structure is the one for oblivious arrays $\widetilde{\mathcal{A}}$.

$\boxed{\Gamma \vdash \tau \rightsquigarrow s}$

$$\text{TR-UnitT}\quad \frac{}{\Gamma \vdash \mathbb{1} \rightsquigarrow 0} \qquad \text{TR-OInt}\quad \frac{}{\Gamma \vdash \widehat{\mathbb{Z}} \rightsquigarrow 1} \qquad \text{TR-OProd}\quad \frac{\Gamma \vdash \tau_1 \rightsquigarrow s_1 \qquad \Gamma \vdash \tau_2 \rightsquigarrow s_2}{\Gamma \vdash \tau_1 \mathbin{\widehat{\times}} \tau_2 \rightsquigarrow s_1 + s_2}$$

$$\text{TR-OSum}\quad \frac{\Gamma \vdash \tau_1 \rightsquigarrow s_1 \qquad \Gamma \vdash \tau_2 \rightsquigarrow s_2}{\Gamma \vdash \tau_1 \mathbin{\widehat{+}} \tau_2 \rightsquigarrow 1 + \max\ s_1\ s_2} \qquad \text{TR-TApp}\quad \frac{}{\Gamma \vdash \widehat{X}\ x \rightsquigarrow \widehat{X}\ x} \qquad \text{TR-LetT}\quad \frac{\Gamma \vdash e \rightsquigarrow_\perp \dot{e} \qquad x{:}_\perp\tau_1, \Gamma \vdash \tau \rightsquigarrow s}{\Gamma \vdash \mathtt{let}\ x{:}_\perp\tau_1 = e\ \mathtt{in}\ \tau \rightsquigarrow \mathtt{let}\ x = \dot{e}\ \mathtt{in}\ s}$$

Fig. 16. Selected rules for translating core Taype oblivious type to sizes in Oil

when its leaky counterpart is used, rather than the label of the whole expression. To see why, recall the typing rule T-IfNoDep from Figure 10: if the label of the discriminee is $\top$, the label of the whole expression must also be $\top$. On the other hand, we do allow the discriminee to be non-leaky, even if the whole expression is leaky. In this case, we simply use the standard `if` statement, as leaks can only occur in a subexpression. A similar strategy applies when translating `case`. The TR-If rule illustrates why we annotate conditionals and case statements with their result type τ: this type is used to resolve the leaky if instances associated with τ via a call to the metafunction $\widehat{\mathtt{if}}$.

Figure 13b presents some of the translation rules for leaky and oblivious constructs. This translation is more involved, as it needs to account for the switch to Oil's oblivious arrays. This is straightforward for simple data types: unit values are simply encoded as an empty array (TR-Unit), while the translation of an oblivious pair simply concatenates the arrays produced by the translation of its two components (TR-OPair). Translating the destructor for oblivious pairs is more interesting (TR-OPCase), as it needs to extract each component from a flat array. To see how this is possible, observe that the "size" of an oblivious value is determined by its type, otherwise we risk leaking private information through this side-channel: thus, we can determine the location of each component of a pair based solely on their types. We do so via an auxiliary relation, $\Gamma \vdash \tau \rightsquigarrow s$, given in Figure 16, which translate Taype types to Oil size expressions.

```
fn inl^ : ℕ → ℕ → 𝒜 → 𝒜 =
  λm n â ⇒
    let payload =
          if n ≤ m then â
          else â ++ 𝒜(n−m)
        tag = s_𝔹 True
    in payload ++ tag
```

Fig. 17. Oblivious injection

The translation of oblivious injections provide another example of how this relation is used. The TR-OInl rule relies on the auxiliary function shown in Figure 17. This function takes as input the sizes of the left and right components and the injection payload, and produces an oblivious array containing the tag and payload, padding it out to the size of the larger component to avoid leaking information through its representation. Somewhat counterintuitively, the tag is placed after the payload for reasons that will be discussed in Section 4.3. For example, the translation of $\widehat{\mathtt{inl}}$<$\widehat{\mathbb{Z}} \mathbin{\widehat{+}} \widehat{\mathbb{Z}} \mathbin{\widehat{\times}} \widehat{\mathbb{Z}}$> `[2]` computes to `[2, 0, 1]`, while $\widehat{\mathtt{inr}}$<$\widehat{\mathbb{Z}} \mathbin{\widehat{+}} \widehat{\mathbb{Z}} \mathbin{\widehat{\times}} \widehat{\mathbb{Z}}$> `⟨[3],[4]⟩` computes to `[3, 4, 0]`.

The remaining rules in the figure adopt similar strategies; relying on a combination of leaky structures to patch up leaky constructs and the size relation to bridge the gap between oblivious types in Taype and oblivious arrays in Oil: the rule for `tape` (TR-Tape), for example, simply delegates the repair to $\widetilde{\mathtt{tape}}$, from Figure 7, which encodes the tape rules S-TapeOIf and S-TapeProm. Similarly, the TR-OCase uses the size to look up the location of the tag of a sum type, before processing both branches with a leaky $\widehat{\mathtt{if}}$ expression which eventually discards the unused branch. Note that the payload x extracted from the injection in the "wrong" branch always uses the right size for that type: when matching on the previous example, x will be `[2, 0]` in the second branch.

The translation of top-level definitions is straightforward; Figure 18 provides the rules for oblivious ADTs and functions of this translation. The elided translation of ADTs simply relies on the generation algorithm from Figure 15. The resolution procedures, $\widehat{\mathtt{if}}$ and `prom`, are also straightforward, available in the appendix.

TR-OADT

$$\frac{\mathtt{x}{:}_{\bot}\tau' \vdash \tau \rightsquigarrow \mathtt{s}}{\mathtt{obliv}\ \widehat{\mathtt{X}}\ (\mathtt{x}{:}\tau') = \tau \rightsquigarrow \mathtt{fn}\ \widehat{\mathtt{X}}{:}\ [\![\tau']\!]_{\bot} \to \mathbb{N} = \lambda \mathtt{x} \Rightarrow \mathtt{s}}$$

TR-FUN

$$\frac{\cdot \vdash \mathtt{e} \rightsquigarrow_{l} \dot{\mathtt{e}}}{\mathtt{fn}\ \mathtt{x}{:}_{l}\tau = \mathtt{e} \rightsquigarrow \mathtt{fn}\ \mathtt{x}{:}[\![\tau]\!]_{l} = \dot{\mathtt{e}}}$$

$\boxed{\mathtt{D} \rightsquigarrow \dot{\mathtt{D}}}$

Fig. 18. Selected rules for translating core TAYPE definitions to OIL definitions

Our translation algorithm is guaranteed to terminate, even when the source program does not. The reason can be seen in our translation rules, as every (mutually) recursive call to the translation judgment is applied to a structurally smaller core TAYPE sub-expression. The algorithm enjoys a stronger totality property: translation of a well-typed core TAYPE program never fails, i.e. a well-typed program satisfies all the side-conditions of the translation rules:

THEOREM 4.1 (TOTALITY OF TRANSLATION). *If* $\Gamma \vdash \mathtt{e} :_{l} \tau$ *and* $\mathtt{e}$ *is in ANF, then there exists an* OIL *expression* $\dot{\mathtt{e}}$ *such that* $\Gamma \vdash \mathtt{e} \rightsquigarrow_{l} \dot{\mathtt{e}}$.

The proof (and an analogous theorem for the type-to-size translation) is given in the appendix.

4.3 Translation for Conceal and Reveal Phases

Secure multiparty computations typically consist of three phases: a *conceal phase*, an *oblivious computation phase*, and a *reveal phase*. In the conceal phase, private data owners "encrypt" and share their data before the core computation takes place, while the oblivious output is revealed to all (or the privileged) parties in the reveal phase. In order to provide a complete solution, we also produce secure implementations of these two phases. Thankfully, section and retraction functions provide templates for concealing and revealing private data. Our toolchain thus translates section and retraction functions to special versions that implement each phase.

Translating the retraction functions needed for the reveal phase is simple: we simply make all the leaky operations "leak" by renaming all leaky operations in a retraction function, and link them to the revealing versions. For example, $\mathtt{s}_{\mathbb{Z}}$ is renamed to $\mathtt{unsafe_s}_{\mathbb{Z}}$. The retraction functions themselves are also renamed so client programs can use them to reveal the results of the computation.

Translating the section functions needed for the conceal phase is more involved. The main problem is that, unlike the core computation, only the private data owner can run the conceal function, as other parties do not have the data. But many MPC protocols, e.g., ones based on secret-sharing, require all parties to help create the encryption of the private information, so this has to be done synchronously. In our setting, since private data is encoded as oblivious arrays, all parties have to encrypt the elements of the same index at the same time. For example, during the conceal phase, if Alice is encrypting the third element of the array, Bob needs to do the same. However, this is not trivial to enforce: how does Bob know which element Alice is currently dealing with, when he does not have the data? Naturally, Bob may only construct the oblivious array from left to right, which means that Alice needs to do the same. Our current implementation thus requires section functions to always construct the array from left to right. Fortunately, the natural encoding strategies for all the oblivious data types we have considered so far satisfy this restriction, with one exception: sum types typically encode the tag before the data, but, by the time the oblivious injection function $\widehat{\mathtt{inl}}$ receives the payload, it has already been "encrypted" under a CBV semantics. To skirt this problem, our solution is to simply have the oblivious injection function encode the payload and then the encrypted tag, as we previously discussed.

5 IMPLEMENTATION

We have implemented the above approach as a compiler that takes as input a TAYPE program containing the functions to be computed (as well as any auxiliary functions), the public views, and section and retraction functions. After type checking these pieces, our toolchain produces

Computation	Alice's input	Bob's input	Description
is-taller	record	record	a variant of the millionaire problem, comparing height
is-obese-by-id	database	ID	whether the record of a given ID is obese (according to BMI)
healthy-rate-by-age	database	a range of ages	the percentage of healthy members of an age range (according to BMI)
min-euclidean-distance	database	record	the minimum euclidean distance between a database and a given record
database-analytics	database	database	calculate the mean and variance of the ages over these databases
mean-squared-error	database	database of BMIs	the mean squared error between the estimated BMIs (from Bob) and the actual BMIs (from Alice); the two databases may not contain the same records, and are matched with IDs (similar to joining tables)
decision-tree	decision tree	record	classify a medical record via a private decision tree
secure-calculator	expression and assignment	expression and assignment	a 2-round arithmetic expression evaluation; each party provides a private arithmetic expression and some private variable assignment depending on the round
voting	tabulated votes	tabulated votes	return the candicate with the most votes
k-means	list of vectors	list of vectors	partition vectors using the k-means clustering algorithm

Fig. 19. Summary of programs used in our case studies

OCaml implementations of the conceal and reveal phases, as well as an OCaml implementation of the multiparty computation, all of which are specialized to the desired public view. We have also implemented two optimizations to improve the performance of generated code: the first `tape`s leaky expressions returning primitive values earlier, to eliminate duplicate $\widehat{\texttt{if}}$ expressions in recursive calls, and the second uses *tupling* [Bird 1984; Chin 1993] to combine calls to section and retraction functions with calls to the size function, in order to eliminate duplicate size computations.[7] The output programs are clients of a module that provides an implementation of Oil's oblivious arrays and oblivious operations. Linking the generated programs with an implementation of this interface, or *driver*, produces a library that a programmer can use to build a secure application: they simply gather the private data, "encrypt" the data using the generated conceal functions, call the multiparty functionality from the library, and finally reveal the result using the generated reveal functions. As the calculator case study in the next section demonstrates, programmers can also implement multi-round computation by chaining together calls to this library.

Our current implementation features two drivers: a plaintext driver and a cryptography-backed driver. The plaintext driver computes its results in the clear, and is intended for testing purposes and for establishing a performance baseline without any cryptographic overhead. The cryptographic driver uses the popular open-source EMP toolkit [Wang et al. 2016] to implement secure computations. This library is based on Yao's Garbled Circuit [Yao 1982] for semi-honest 2-party MPC. Integrating a new backend into our framework is conceptually simple: the driver just needs to implement an interface consisting of oblivious integer encryption, decryption and its arithmetic. Our EMP toolkit backend consists of boilerplate code for FFI (foreign function interface), for example. Other aspects of the driver, such as array operations, are independent of the cryptographic backends, and can thus be shared among all drivers.

6 EXPERIMENTS

Our experiments consist of a set of case studies that showcase the applicability of our approach, summarized in Figure 19, and a set of micro-benchmarks that examine the empirical benefits of being able to trade off security for performance.

6.1 Case Study: Medical Records

Our first collection of case studies are inspired by problems in the healthcare setting, where legal and privacy concerns keep parties from freely sharing their data. These benchmarks use a variety of data structures: patient data is represented as a record with fields for a patient's ID, age, height and weight, a database is encoded as a list of patient records, and a classifier is implemented as

[7] The appendix provides the details of both optimizations.

a decision tree over health data. The oblivious types for these data structures admit interesting public views: a particular health record may choose to keep either its ID, or medical data (height and weight) secret, as in Figure 3; a database adopts the privacy settings (i.e. revealing either ID or medical data) of its individual records, and a decision tree obscures the threshold that a feature is compared against at a given node, but not the overall structure. Using these representations, we have implemented a number of secure computations: biometric matching (minimum euclidean distance) between a single record and a database, calculating the percentage of healthy members of an age group according to the Body Mass Index (BMI), calculating statistics such as mean and variance over multiple private databases, and classifying a patient record using a private decision tree. Figure 19 includes each of these programs. Notably, the computation in each of these benchmarks was written without a privacy policy in mind; instead, our compiler took care of enforcing each policy, as encoded by an oblivious type and section and retraction functions.

6.2 Case Study: Secure Calculator

To showcase our support for computations involving richly structured data, we have implemented a secure interpreter for a simple arithmetic expression language. In this case study, each user provides a private expression and an assignment to some variables. The result is securely computed by evaluating the first party's expression using the second party's assignment; the result of this expression is then used to evaluate the second party's expression, along with the first party's private value, as shown in Figure 20. Not only does this case study use a rich data structure for expression, it also shows that we can readily compose the generated library functions to implement a more complex workflow, such as a multi-round computation.

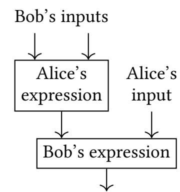


Fig. 20. Workflow of the secure calculator

Discussion. As mentioned in Section 1, in existing frameworks, it is the program's responsibility to enforce the privacy policy. In contrast, our medical records and secure calculator case studies show that, in our framework secure functionality can be written in a conventional functional language, agnostic to a particular privacy policies. On the other hand, implementing oblivious types, section and retraction functions is analogous to other common programming tasks: an oblivious type is essentially a different representation of the underlying data type, while section and retraction functions are effectively conversion functions between oblivious and public types. Importantly, our abstraction allows programmers to write all these "boilerplate" functions once and for all, regardless of a particular target computation.

6.3 Micro-benchmarks

To evaluate the performance of our compiler,[8] we have built a number of micro-benchmarks that showcase the performance tradeoffs between privacy and performance. Our first micro-benchmark is a standard classification scenario, where one party wants to classify their private data using a decision tree belonging to another party [Kiss et al. 2019; Malik et al. 2021; Wu et al. 2016]. The data being classified is given as a tuple with eight private integers as features. This experiment considers 4 public views for the decision tree: maximum height, the spine, spine including the feature index of each node, and the whole tree. Note that this last view is not unrealistic: in outsourced secure computation, such as FHE [Gentry 2009], the decision tree owner may perform all computation, independent of the other party. In this scenario, the whole tree can be revealed because the computing party owns it, but the computation should not reveal any information about

[8] All results are averaged across 10 runs, on an M1 MacBook Pro with 16 GB memory. All parties run on the same host with local network communication.

```
// The payload of Leaf is a decision, and that
// of Node is a feature index and a threshold.
data tree = Leaf ℤ | Node ℤ ℤ tree tree
obliv tree_all (_ : tree) = 𝟙
```

(a) All information

```
obliv tree_max (k : ℤ) =
  if k ≡ 0 then ℤ̂
  else ℤ̂ +̂
       ℤ̂ ×̂ ℤ̂ ×̂ tree_max (k−1) ×̂ tree_max (k−1)
```

(b) Maximum height

```
data spine = SLeaf | SNode spine spine
obliv tree_spine (s : spine) =
  case s of SLeaf ⇒ ℤ̂
  | SNode l r ⇒
    ℤ̂ ×̂ ℤ̂ ×̂ tree_spine l ×̂ tree_spine r
```

(c) Tree spine

```
data spineF = SFLeaf | SFNode ℤ spineF spineF
obliv tree_spineF (s : spineF) =
  case s of SFLeaf ⇒ ℤ̂
  | SFNode _ l r ⇒
    ℤ̂ ×̂ tree_spineF l ×̂ tree_spineF r
```

(d) Tree spine with feature indices

Fig. 21. Definitions of oblivious decision trees with different public views

the other party's data. The definition of the decision tree is shown in Figure 21a, together with an "oblivious" version that simply reveals the whole tree. Figure 21 also includes views for three other policies. The section and retraction functions for each view are analogous to those in Figure 5. For each public view, we test on a small tree of depth 1, and other trees of depth 16. A *full tree* has exponentially many nodes, while an *eighth sparse tree* has roughly 1/8 of the nodes in a full tree, and a *very sparse tree* has only 16 nodes.

Figure 22 reports the performance impact of each view on the total run time. The results are as expected: revealing the whole tree results in the best performance, while sharing only the maximum height is quite slow. In the case of maximum height, the number of nodes in the actual decision tree does not affect the performance, as the structure of the tree is kept secret. Knowing both the spine and the feature index of each node improves performance, compared to knowing only the spine, as the computation does not need to obscure which feature is used at each decision point. When the underlying decision tree is relatively full, leaking more information about its structure does not improve performance, as the program does not need to perform wasted computation to ensure a constant time algorithm. Indeed, the fuller the private tree is, the less is gained by a more permissive public view. Of course, the owner of the tree must ultimately decide if they are willing to reveal how the tree is close to this worst case scenario. Again, the decision algorithm is agnostic of the actual public views, allowing for swapping privacy policies without any changes to the program logic.

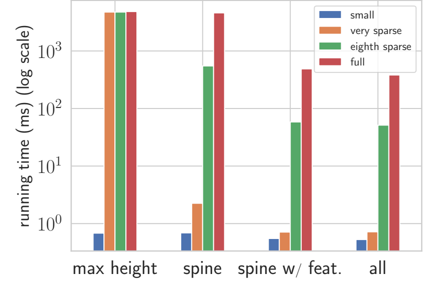
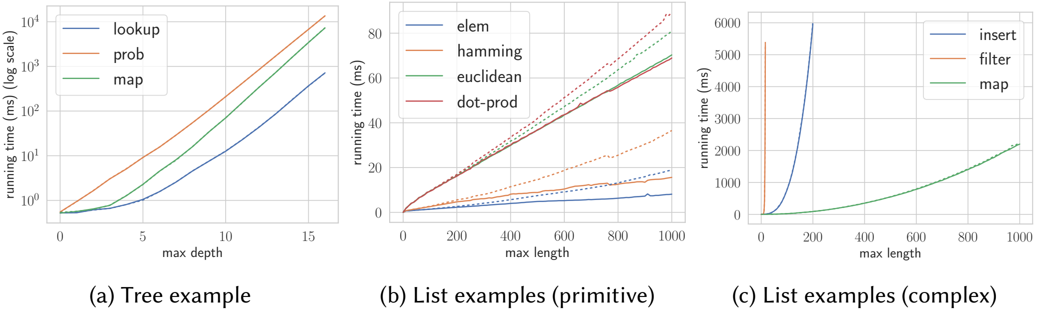


Fig. 22. Decision tree

We also evaluate the performance of a set of standard operations on trees, using its maximum height as the public view. These benchmarks consist of a membership test, computing the probability of an event given a probability tree diagram, and a map function that adds a private integer to each node in a tree. Figure 23a presents the performance results for these benchmarks. Despite the inherently expensive cryptography required by the conservative public view, all the benchmarks finish in under 15 seconds.

Finally, we have implemented a similar set of micro-benchmarks for oblivious lists, using the length of the list as the public view. We subdivide these benchmarks into those that return a primitive value (i.e. an integer or boolean), and those that return an oblivious list. Even though the returned lists often reveal too much about the private input, they could be useful as an intermediate result of a bigger computation or as an input to the next round of computation. Figure 23b presents



(a) Tree example (b) List examples (primitive) (c) List examples (complex)

Fig. 23. Micro-benchmarks

the performance for the first category, which includes a membership check, computing of the hamming and euclidean distances between two lists, and taking the dot product of two lists. All of these examples are amenable to the optimization mentioned in the previous section, resulting in reasonable running times. We use a dotted line for results without our tupling optimization, and a solid line for when the optimization is enabled. With tupling is used, their performance is linear in the size of the input list (as it is in the insecure setting). The second category includes insertion into a sorted list, and two higher-order examples: mapping a function that adds a private integer to all the elements of a list, and a filter function that drops all the elements greater than a private integer. Since these examples do not return primitive values, the early-tape optimization does not apply, resulting in slower performance, as Figure 23c shows. The tupling optimization does not have much impact, as its gains are overshadowed by the complexity of having to delay repairs to the leaky result values.

7 RELATED WORK

Yao [1982] was the first to formally introduce secure multiparty computation. Solutions based on cryptography-backed protocols roughly fall into two categories [Evans et al. 2018; Hazay and Lindell 2010]: those based on secret-sharing [Beimel 2011; Goldreich et al. 1987; Maurer 2006] including Yao's Garbled Circuits, and those based on homomorphic encryption [Acar et al. 2018; Gentry 2009]. Alternative solutions based on virtualization [Barthe et al. 2014, 2019] or secure processors [Hoekstra 2015] have also been proposed. Our implementation uses the EMP toolkit [Wang et al. 2016] for its secure backend, but is compatible with other solutions based on cryptographic protocols under the mild requirement that they implement primitives for secure integer operations.

The type system of TAYPE is an example of a *security-type system* [Sabelfeld and Myers 2003; Zdancewic 2002], a type-based approach for information flow control. Similar to the security labels often used in these systems, our leakage labels are used to track if a term is leaky and kinds keep track of whether a term is oblivious. TAYPE's type system is similar to $\lambda_{\text{OADT}\textbf{+}}$'s, but it requires explicit label promotion and uses uniform labelling of subexpressions, to aid in translation. TAYPE furthermore uses a bidirectional typing algorithm to implement its dependent type checker. The type system of TAYPE guarantees that there are no timing channels in well-typed programs, similar to the security-type systems of other *constant-time* languages [Cauligi et al. 2019].

Our obliviousness guarantee is a strengthened variant of *memory trace obliviousness* (MTO) [Liu et al. 2013], which itself provides stronger guarantees than most information flow type systems. Under MTO, the pattern of memory access generated by a program are required to be indistinguishable, in addition to its output. This work also proposed a language based on *Oblivious RAM* [Goldreich 1987; Goldreich and Ostrovsky 1996; Stefanov et al. 2013] and transformation techniques to ensure this property. However, this threat model is weaker than that of TAYPE. On

the one hand, it does not consider timing channel: while memory access traces include instruction fetches, which ensures the branches of a secure conditional always run the same number of instructions, the instructions themselves can still exhibit different timing behaviors. For example, the program `if s > 0 then s := p + p else s := p * p` is secure in their model, as both branches produce the same memory access pattern (including instruction fetches), but the second branch is slower, assuming multiplication is slower than addition in the CPU. On the other hand, under MTO, adversaries cannot observe the instructions executed by the CPU. This is not the case in the MPC setting (especially in secret-sharing based schemes), as every party is a potential adversary that can observe instructions: `if s > 0 then s := p + 1 else s := p + 2` is accepted in their model, but an adversary in our model is able to discern if the program is computing `p+1` or `p+2`, even if they have the same timing behavior. In contrast, the traces we consider include every program state under a small-step semantics, which rules out these two examples.

Several high-level languages have been proposed to help programmers write secure multiparty computations [Hastings et al. 2019]. In contrast to Taype, most of these languages either do not support recursive data types at all, or require any structural information to be public (Obliv-C [Zahur and Evans 2015] and ObliVM [Liu et al. 2015]). To the best of our knowledge, none of these languages decouple security policies and program logic, as Taype does. On the other hand, many of these languages focus on different aspects of oblivious computation that we do not consider. Wysteria and Wys* [Rastogi et al. 2014, 2019] are functional languages that focus on *mixed-mode computation*. Symphony [Sweet et al. 2023] is a successor of Wysteria which provides first class support for coordinating parties, allowing for more reactive applications. λ_{obliv} [Darais et al. 2020] is a functional language for writing probabilistic programs, and can be used to implement oblivious cryptographic algorithms, such as Oblivious RAM.

Several projects have focused on improving the performance of secure applications by intelligently selecting the most efficient combination of protocols for a particular computation. Viaduct [Acay et al. 2021], for example, optimizes high-level secure programs by transforming them into distributed programs, and automatically choosing the most efficient protocols for each subcomputation. HyCC [Büscher et al. 2018], ABY [Demmler et al. 2015] and MOTION [Braun et al. 2022] are similar frameworks for enabling *mixed-protocol computation*. The HACCLE [Bao et al. 2021] toolchain is a multi-stage compiler for optimizing circuit generation.

8 CONCLUSION

Secure multiparty computation enables different parties to compute functions over private data without leaking extra information, but writing these applications remains challenging. Existing high-level MPC languages require programs to explicitly enforce privacy policies, making it difficult to update policies and to explore tradeoffs between privacy guarantees and performance. This paper presented Taype, a language for secure multiparty applications that decouples these concerns. Our experiments feature a diverse set of benchmarks that were written without security policies in mind, and a wide range of security policies that went beyond whether a particular field is "secret or not". Our results demonstrate the performance benefits that can result from being able to easily trade off privacy for performance.

ACKNOWLEDGMENTS

We thank Patrick LaFontaine, Robert Dickerson, our shepherd Pierre Geneves, and the anonymous reviewers for their detailed comments and suggestions. We also thank Kirshanthan Sundararajah, Raghav Malik, and Milind Kulkarni for their stimulating discussions, and Jianfei Gao for his help with plotting benchmark figures. This material is based upon work partially supported by Cisco Systems under award #23013611, and IARPA under contract #2019-19020700004.

9 SOFTWARE AVAILABILITY

An artifact containing our implementation of Taype, its source code, and the source for all the benchmarks in our experiments with instructions is publicly available [Ye and Delaware 2023]. The appendix is included in the auxiliary material.

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, 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

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

R. S. Bird. 1984. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica* 21, 3 (Oct. 1984), 239–250. https://doi.org/10.1007/BF00264249

Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. 2022. MOTION - A Framework for Mixed-Protocol Multi-Party Computation. *ACM Transactions on Privacy and Security* 25, 2 (March 2022), 8:1–8:35. https://doi.org/10.1145/3490390

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

Wei-Ngan Chin. 1993. Towards an Automated Tupling Strategy. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '93)*. Association for Computing Machinery, New York, NY, USA, 119–132. https://doi.org/10.1145/154630.154643

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

Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2015.23113

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (may 2021), 38 pages. https://doi.org/10.1145/3450952

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

O. Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87)*. Association for Computing Machinery, New York, NY, USA, 182–194. https://doi.org/10.1145/28395.28416

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

Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (May 1996), 431–473. https://doi.org/10.1145/233551.233553

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

Ágnes Kiss, Masoud Naderpour, Jian Liu, N. Asokan, and Thomas Schneider. 2019. SoK: Modular and Efficient Private Decision Tree Evaluation. *Proceedings on Privacy Enhancing Technologies* 2019, 2 (April 2019), 187–208. https://doi.org/10.2478/popets-2019-0026

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, 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

Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. Association for Computing Machinery, New York, NY, USA, 299–310. https://doi.org/10.1145/2508859.2516660

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

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

David J. Wu, Tony Feng, Michael Naehrig, and Kristin Lauter. 2016. Privately Evaluating Decision Trees and Random Forests. *Proceedings on Privacy Enhancing Technologies* 2016, 4 (Oct. 2016), 335–355. https://doi.org/10.1515/popets-2016-0043

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: PLDI23 Artifact. https://doi.org/10.5281/zenodo.7806981

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.

Received 2022-11-10; accepted 2023-03-31