Oblivious Algebraic Data Types

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

Secure computation allows multiple parties to compute joint functions over private data without leaking any sensitive data, typically using powerful cryptographic techniques. Writing secure applications using these techniques directly can be challenging, resulting in the development of several programming languages and compilers that aim to make secure computation accessible. Unfortunately, many of these languages either lack or have limited support for rich recursive data structures, like trees. In this paper, we propose a novel representation of structured data types, which we call oblivious algebraic data types, and a language for writing secure computations using them. This language combines dependent types with constructs for oblivious computation, and provides a security-type system which ensures that adversaries can learn nothing more than the result of a computation. Using this language, authors can write a single function over private data, and then easily build an equivalent secure computation according to a desired public view of their data.

CCS Concepts:
• Software and its engineering → General programming languages;
• Social and professional topics → History of programming languages.

ACM Reference Format:

1 INTRODUCTION

It is often the case that the owners of some private data want to compute some joint function of their data: a group of hospitals, for example, may want to calculate some statistics about their patients. In the case that this data is sensitive, the parties may not want (or be legally allowed) to simply pool their data and compute the result. Secure computation provides a solution in such scenarios, allowing multiple parties to perform a joint computation while keeping their sensitive data secure. Secure computation was formally introduced by Yao [1982] in the early 1980s, and has since found many privacy-focused applications, including secure auctions, voting, and privacy-preserving machine learning [Evans et al. 2018; Hastings et al. 2019; Laud and Kamm 2015]. There are two major paradigms for secure computation: secure multiparty computation (MPC), wherein the computation is performed jointly by all parties involved; and outsourced computation, where a computationally powerful entity such as an untrusted cloud provider carries out the computation [Evans et al. 2018]. MPC is typically implemented using cryptography-based protocols, such as Yao’s Garbled Circuits [Yao 1982] or secret-sharing [Goldreich et al. 1987], while outsourced computation can be implemented using a variety of mechanisms, including cryptography-based fully homomorphic encryption [Gentry 2009], virtualization [Barthe et al. 2014, 2019b] and secure processors [Hoekstra 2015].

Writing secure applications which directly use these techniques can be quite challenging and error-prone, however, even if the author has the requisite cryptographic expertise. Thus, starting with Fairplay, the first publicly available MPC compiler [Malkhi et al. 2004], several high-level programming languages and compilers have been proposed to make secure computation accessible to non-experts [Hastings et al. 2019]. For example, Obliv-C [Zahur and Evans 2015] is a C-like language for MPC applications which compiles down to Yao’s Garbled Circuits. Other notable
languages include ObliVM [Liu et al. 2015], Wysteria/Wys* [Rastogi et al. 2014, 2019], $\lambda_{obliv}$ [Darais et al. 2020].

Unfortunately, many of these languages either lack or have limited support for rich recursive data structures, like trees. When such data structures are supported, they typically require leaking information about the structure of the data: in Obliv-C, for example, users can define trees with secure nodes using pointers, but the “shape” of the underlying tree will always be visible to adversaries, as Obliv-C pointers are public data. In this paper, we propose a language that supports algebraic data structures capable of hiding not only their secure payload, but also their own structure. In this system, adversaries are not able to infer, say, whether an oblivious tree is left-heavy or right-heavy by observing the data structures themselves or how they are used.

One major obstacle to securely implementing such data structures is the possibility of timing channels in the programs that use them: the run time of any terminating computation reveals some approximate information about the “size” of the data structures it uses. Authors of secure computations must be careful to not inadvertently reveal more information through such timing side-channels. As an example, consider the following Obliv-C program, which traverses an oblivious array $a$:

```c
for (i = 0; i < MAX_BOUND; i++) {
    // some secure computation on a[i]
}
```

Here, the author has chosen to avoid timing-channels by using a upper bound, MAX_BOUND, on the length of $a$. In effect, MAX_BOUND provides a sort of public view on the structure of $a$, which is then used to ensure a consistent running time for the for loop. Of course, the author also must ensure a is been padded out to this maximum bound and that there are no break statements that depend on the contents of $a$ in the body of the loop. In order to be secure, a computation over structured data types must be carried out without revealing any information outside of this public view, including the structure of the private data.

While this trick to make programs constant-time [Barthe et al. 2014] is easy to implement for computations over simple data types like arrays, it becomes more complicated for richer data structures. First, users have to decide how to manually “pad” data structures so they are consistent with their public view. Second, programmers also have to track the public view throughout the program, making sure it remains consistent throughout. Lastly, richer data structures can have multiple public views, representing different tradeoffs between privacy and performance. As an example, the public views of an oblivious tree include both its maximum depth and its spine. If the spine is known, less padding (and thus less “wasted” computation) is needed. Since the public view, like MAX_BOUND above, directly affects how programs are written, programmers are forced to write different versions of the same program even though the high-level program logic is exactly the same. Ideally, these concerns should be separated, allowing programmers to write the program once and for all, and then select the right public view for their security and performance requirements.

In this paper, we propose a novel representation of structured data types, which we call oblivious algebraic data types (OADTs). Our solution combines dependent types with language constructs for oblivious computation, and a security-type system which ensures that adversaries learn nothing more than the output of the function and the input they provide. Using our language, clients can write a single function over private data, and then build an equivalent oblivious computation over some public view of that data. By switching views, users can explicitly trade off between how much information is leaked via public channels and the performance of the underlying computation.

In summary, this paper presents the following contributions:
• We observe that public views of private ADTs can be naturally expressed using dependent
types with large elimination, allowing for a clean specification of what information is released
at runtime.
• Exploiting this observation, we develop \( \lambda_{OADT} \), a core calculus for writing oblivious programs
using OADTs, whose strong type system ensures computations are secure.
• To enable both a more pleasant programming experience and more modular programs, we
develop an extension of \( \lambda_{OADT} \), dubbed \( \lambda_{OADT} \), which is equipped with a novel semantics
which enable creating, from one single public program, oblivious programs with different
public views.

Both \( \lambda_{OADT} \) and \( \lambda_{OADT} \), and their metatheory have been mechanically formalized in the Coq proof
assistant. These developments can be found in the supplementary material.

2 OVERVIEW

data tree := leaf | node Z tree tree

def lookup (x : Z) (t : tree) : B :=
  case t of
  leaf ⇒ false
  node y tl tr ⇒
    if x ≤ y
    then if y ≤ x
    then true
    else lookup x tl
  else lookup x tr
Fig. 1. Lookup element in a search tree

To illustrate our approach, consider the simple function
in Figure 1, which looks for an element in a search tree by
recursing over the tree. Suppose that Alice, the owner of
a search tree, and Bob, the owner of some integer, want to
check whether Bob’s integer is a member of Alice’s tree,
without revealing any information to each other beyond
what each can learn from their private data and the output.

We adopt a variation of the standard semi-honest threat
model from multiparty computation, where an untrusted
party can observe every intermediate execution step of the
program under a small-step operational semantics. This
threat model reflects standard MPC protocols based on si-
multaneous execution of the program, e.g. those based on
secret-sharing [Beimel 2011; Goldreich et al. 1987], and
outsourced computation where the untrusted evaluators
perform the execution, e.g., those based on fully homo-
morphic encryption [Gentry 2009]. This threat model also
naturally covers weaker adversaries, such as those who can only observe the timing behavior.

Under this threat model, Bob can glean information about Alice’s tree just by examining how
it effects the control flow of the program, even if the tree is perfectly obfuscated. To see how,
consider the execution trace of lookup [4] [t] shown in Figure 2, where [t] is the tree shown in
the first row. The first column of each subsequent row shows the current execution execution step, while

Fig. 2. Execution trace of lookup [4] [t]. The columns show the current state of the program, and information learned by the
owner of the lookup key, respectively.
1:4 Qianchuan Ye and Benjamin Delaware

obliv tree (k : N) :=
if k = 0
then 1
else 1 + \mathbb{Z} \times \text{tree} (k-1) \times \text{tree} (k-1)

obliv tree' (s : spine) :=
\begin{align*}
\text{case } s \text{ of} \\
\text{sleaf } & \Rightarrow 1 \\
\text{snode } & \Rightarrow 1 + \mathbb{Z} \times \text{tree}' s_l \times \text{tree}' s_r
\end{align*}

(a) Maximum depth as public view
(b) Upper bound of spine as public view

Fig. 3. Oblivious trees

the second column shows what Bob can infer at that step. We use square brackets to denote that [t] is an oblivious value, i.e. it cannot be directly observed by a party. At each recursive call to lookup, there are two points that depend on the structure of the tree: the case statement that checks whether to recurse, and the if statement that decides which subtree to recurse on. As the fourth row illustrates, the branch case takes reveals some information about the structure of the current tree (it is non-empty) to Bob. The fifth and sixth row of the figure similarly show how the if statement reveals information about the relationship of the key to the value in the current node. By examining the program immediately following each such test, Bob adds to his knowledge of Alice’s tree. At the end, Bob learns not only output of the function, but also a partial view of the tree’s structure (including the exact node 4 is stored in); this view could be further refined by subsequent lookup operations.

Note that the participants of any terminating multiparty computation have to agree to share some public information: intuitively, simply knowing the number of intermediate steps in an execution of lookup leaks some upper bound on the number of nodes in the tree. Once that concession is made, the choice becomes what information to share– maybe the owner of the tree is okay with sharing its spine, but not the values stored in its internal nodes, or perhaps with revealing some upper bound on its depth\(^1\). The goal then, is to enable parties to compute functions over private data in a way that only depends on some mutually agreed upon public view of that data.

Oblivious Algebraic Data Types. The first component to our solution is our representation of both private data and its corresponding public view using oblivious algebraic data types. Figure 3a gives an example of an oblivious tree whose public view is its maximum depth. The private part of this type is built up from a set of oblivious type formers, e.g. \(\mathbb{Z}\) and \(\uparrow\) are the formers for oblivious fixed-width integers and oblivious coproducts, respectively. Section 3.4 formalizes oblivious data values, but the high-level intuition is that an observer of an execution trace cannot distinguish between the values of an oblivious type. When examining the trace in Figure 2, the oblivious integer \([4]\) is indistinguishable from \([12]\), for example. The key idea behind oblivious ADTs is to use large elimination on the public view to construct a representation of private data. For example, all oblivious trees with a maximum depth of two are represented as:

\[
\text{tree } 2 \equiv 1 + \mathbb{Z} \times (1 + \mathbb{Z} \times 1 \times 1) \times (1 + \mathbb{Z} \times 1 \times 1)
\]

This type roughly corresponds to the tree shown in Figure 4. Every tree of this type is padded to depth 2, even a single "leaf", to avoid leaking structural information. This padding is implied by the

\[\text{1In the case the owner is okay with sharing the entire tree, the computation becomes quite efficient indeed!}\]

use of oblivious coproduct \( \tilde{+} \), as the left injection and the right injection of an oblivious sum will be indistinguishable. The adversaries can not tell them apart by inspecting the payload, even if the two components have different types. The “tag” of a sum value is of course obfuscated as well. Constructing oblivious data types in this way ensures that all private values corresponding to a particular view are indistinguishable to an attacker.

Figure 3b shows the type of oblivious trees using an upper bound on its spine as the public view, where the spine is another user-defined ADT. This definition releases more public information than the one in Figure 3a, but it also enjoys a more efficient representation, as it requires less padding than a complete tree.

An Oblivious Language. Oblivious ADTs are only half the solution to secure computation— it still remains to ensure computations over private values are also oblivious. Even if an attacker cannot tell which values are being compared in \( \text{if} \ [4] \leq [12] \ldots \), they can still learn something about their relationship just by knowing the expression it steps to, as we saw in our previous example. To prevent these sorts of information leaks, we have designed \( \lambda_{OADT} \), a pure functional language for writing secure computations over OADTs. \( \lambda_{OADT} \) is equipped with dependent types with large elimination to express OADTs, and type-based information flow control to guarantee oblivious computations.

Key to this calculus are its operations for securely constructing and destructing oblivious data values. As an example of these operations, consider the following \( \lambda_{OADT} \) expression, which compares two secure integers to determine what value to return:

\[
\text{mux} \ (\text{[3]} \leq \text{[4]} \ (\text{[5]} \ \tilde{+} \ [1]) \ (\text{[6]} \ \tilde{+} \ [1]))
\]

We use ˆ to denote secure versions of standard operations, such that \( [4] \tilde{+} [3] \rightarrow [7] \) and \( [4] \leq [3] \rightarrow [\text{false}] \). Here, \( \text{mux} \) is a special conditional which returns an oblivious value according to the value of an oblivious boolean. In order to avoid leaking information, \( \text{mux} \) generates the same evaluation trace regardless of the value of the discriminate. To do so, it fully evaluates both branches before stepping to the final (oblivious) result:

\[
\text{mux} \ (\text{[3]} \leq \text{[4]} \ (\text{[5]} \ \tilde{+} \ [1]) \ (\text{[6]} \ \tilde{+} \ [1]) \rightarrow \text{mux} \ [\text{true}] \ [6] \rightarrow [6]
\]

Replacing \( [3] \) with \( [5] \) in the initial expression yields the same execution trace, modulo the private values at each step. Thus, nothing about the private information can be inferred by observing the execution:

\[
\text{mux} \ (\text{[5]} \leq \text{[4]} \ (\text{[5]} \ \tilde{+} \ [1]) \ (\text{[6]} \ \tilde{+} \ [1]) \rightarrow \text{mux} \ [\text{false}] \ [6] \rightarrow [7]
\]

The oblivious case statement behaves similarly, with the additional wrinkle that the pattern variables of the “wrong” branch are bound to some arbitrary oblivious values, which Section 3 explains in full detail.

\( \lambda_{OADT} \) is equipped with a security-type system [Sabelfeld and Myers 2003], to ensure the correct use of its secure operations. The full details of this type system can be found in Section 3.3, but at a high-level it enforces three key policies. First, oblivious types can only be built from oblivious types. For example, a oblivious coproduct cannot be built from public types, such as \( Z \tilde{+} Z \). If this were allowed, an adversary could infer whether a value of this type is a left or a right injection by observing the payload. Second, secure operations like \( \text{mux} \) can only be applied to oblivious terms. \( \text{mux}[\text{true}] \ 1 \ 2 \) is prohibited, for example, as knowing the public result of this \( \text{mux} \) reveals the oblivious discriminate. Third, types are treated as public information, otherwise the parties could not even agree on the data representation. Thus, oblivious types can only depend on public types:

\[
\text{mux} \ [\text{true}] \ Z \ Z \text{ is not a valid type in } \lambda_{OADT}.
\]

Figure 5 presents an oblivious implementation of the \( \text{lookup} \) function for the oblivious tree from Figure 3a. While the high-level program logic is the same, extra care is needed to ensure correct use
of the oblivious tree. First, the function takes an extra argument for the public view; the argument needs to be correctly passed to every recursive call. Second, the function eliminates the public view, following the definition of \texttt{tree}, before accessing any secure data. Third, public constants and operations are replaced by their secure counterparts: e.g., \texttt{if} is replaced by \texttt{mux}. Similarly, the constants \texttt{true} and \texttt{false} are wrapped by the \texttt{sB} operation, which acts like a coercion from public Booleans to oblivious Booleans.

\textbf{A More Ergonomic Oblivious Language.} While the implementation of \texttt{lookup0} is guaranteed to be secure, it is quite far from the "standard" implementation of \texttt{lookup} in Figure 1, as its control flow has been restructured to only depend on public inputs and to meet the demands of a secure type system. As a consequence, a programmer must write distinct versions of \texttt{lookup} for each public view, despite the fact that the high-level program logic is exactly the same. Note that \texttt{lookup} is, in fact, a valid \texttt{OADT} program, \textit{as long as} it is applied to public data. This observation suggests the implementation of \texttt{lookup} sketched to the right in Figure 6, which simply converts its private inputs to public versions, applies \texttt{lookup} to those arguments, and converts the result back to an oblivious value. From a cryptographic perspective, the arrow labeled \texttt{rZ} corresponds to decrypting the secure inputs, while the arrow labeled \texttt{sB} corresponds to encryption. We refer to these two conversion functions as \texttt{retraction} and \texttt{section} to reflect their desired relationship: a retraction is a left inverse for section. There is a fundamental flaw with this approach, however: applying a retraction in this manner completely leaks the private inputs of \texttt{lookup}! Thus, this program must be rejected by \texttt{OADT}'s type system as insecure.

The ideal language for oblivious computation would permit implementations that combine the clarity of \texttt{lookup} with the security guarantees of \texttt{lookup0}. In pursuit of this goal, we have developed an extension of \texttt{OADT}, called \texttt{OADT\_\*}, that allows implementations that follow the recipe sketched in Figure 6 \textit{without} compromising obliviousness. Our key idea is to have the semantics of \texttt{OADT\_\*} repair or "tape up" potentially leaky expressions during execution. This allows users to write section and leaky retraction functions that convert between oblivious and public values, relying on the semantics to ensure oblivious execution of any program that uses those functions.

To understand how this works, consider the execution trace of the simple \texttt{OADT\_\*} program shown in Figure 7a. The new conditional \texttt{if} is similar to \texttt{mux} in \texttt{OADT}, but it allows non-oblivious branches. Note that this \texttt{if} would leak the value of its private condition if it was evaluated using the semantics of \texttt{mux}. Similar leaks occur for any \texttt{if} expression whose branches can evaluate to a public value. The idea behind the semantics of \texttt{OADT\_\*} is straightforward: since \texttt{if} only leaks information when it is evaluated, we will simply not do that! Rather, the surrounding \texttt{tape} annotation tells \texttt{OADT\_\*} to defer reducing \texttt{if} \textit{until it is safe to do so}. This example makes progress by distributing the surrounding \texttt{if}.
statement into its branches and then evaluating both branches to oblivious values instead. Once both branches of an `if` are evaluated to oblivious values, it can be securely reduced to a `mux` to produce the final result. Note that swapping `[true]` with `[false]` in this example produces the exact same trace, modulo oblivious values.

This example demonstrates the two key idea behind the semantics of \( \lambda \text{OADT} \): avoid leaks by delaying evaluation of potentially insecure expressions, while still making progress by distributing the surrounding context into such expressions. This strategy works for contexts like function application as well, as the example in Figure 7b shows. Figure 7c includes an example of a potential leak of oblivious integers via the \( r_Z \) operation, a leak that is ultimately patched using \( s_Z \). The program first progresses by distributing the insecure addition operation into retraction, then obliviously adding the result. After evaluating the oblivious addition, we have \([5]\), and \( s_Z \) and \( r_Z \) can “cancel” each other, as the functions are effectively inverses. As \([5]\) is already an oblivious value, `tape` becomes a no-op in this example.

Figure 8 shows the section and retraction functions for `tree`, along with a version of `lookup` implemented using the recipe from Figure 6. While the section function is not used in `lookup`, it is needed for functions that return an oblivious tree. Function definitions in \( \lambda \text{OADT} \) require an additional annotation which signals if the function body includes any potentially leaky operations (e.g. `if`) that needs to be patched by the context surrounding the function call. Section 4 discusses how the type system of \( \lambda \text{OADT} \) uses these annotations in more detail. At a high level, its type system enforces two polices. First, as types are always public, they should not contain any potential leaks: any type which depends on `if [true] 1 2` is disallowed, for example. Next, because only terms that
evaluate to oblivious values can be patched up, our type system ensures that terms with potential
leaks, e.g. a call to a retraction function or an \(\text{if}\), are obliviously typed.

Having completed the overview of our solution, we now present a detailed accounting of our
calculi for oblivious computation.

3 \(\lambda\text{OADT}, \text{FORMALLY}\)

This section formalizes \(\lambda\text{OADT}\), a core calculus for programming with OADTs. The calculi described
in this section and the next have been mechanized in the Coq proof assistant; both developments
can be found in the supplementary material.

3.1 Syntax

The core syntax of \(\lambda\text{OADT}\) is shown in Figure 9. The full language, which can be found in the
supplementary material, also includes let bindings. For simplicity, the core calculus of \(\lambda\text{OADT}\) does
not include primitive fixed-width integers. We discuss how the language may be extended with
primitive integers in Section 4.5. As \(\lambda\text{OADT}\) is dependently typed, types and terms belong to the
same syntactic class, although by convention, we use the metavariable \(\tau\) to refer to types, and \(e\)
to terms. \(\lambda\text{OADT}\) programs consist of an expression and a global context of public ADTs, oblivious
ADTs, and functions. These are defined using \text{def}, \text{data}, and \text{obliv}, respectively. Using a global set of
function definitions naturally supports general recursion and mutual recursion. When possible, we
use lower case \(x\) for function names, upper case \(X\) for public ADT names, and \(b\) for the names of
oblivious ADTs.

Types in \(\lambda\text{OADT}\) include dependent function types \((\Pi)\), sums \(+\), products \(\times\), and booleans \(\text{B}\); as
well as oblivious sums \(\text{b}+)\) and booleans \(\text{bB}\). In \(\lambda\text{OADT}\), the typing rules and semantics for oblivious
types are quite different from their public counterparts, which is why we chose to assign them
distinct syntax, as opposed to using security labels [Zdancewic 2002]. \(\lambda\text{OADT}\) supports type-level
computation, allowing users to define types using application, \text{if}, and \text{case}. Sum and product types
are also allowed to have both oblivious and public components, allowing types to have both public
and oblivious parts.
### 3.2 Semantics

Figure 10 defines a relation for the small-step operational semantics of $\lambda_{OADT}$. The judgements of this relation have the form $\Sigma \vdash e \rightarrow e'$, and are read as “$e$ steps to $e'$ under the global context $\Sigma$”. Since a $\lambda_{OADT}$ program is evaluated under a fixed global context, we often abbreviate this judgement as $e \rightarrow e'$, referring to $\Sigma$ only when needed. The S-Contr rule uses the evaluation contexts ($E$) defined at the bottom of Figure 10 to evaluate subexpressions. While these evaluation contexts are...
not inductively defined, it is possible to recursively apply S-Ctx when evaluating subtrees. This formulation is more convenient for the \( \lambda_{OADT} \) formalization in the next section.

Most of the non-oblivious rules are standard, although application is evaluated from right to left so that it can accomodate both oblivious ADTs and functions. To keep the semantics succinct, several of the rules use the \( \text{ite} \) meta-function, which returns \( e_1 \) when its first argument is true, and \( e_2 \) otherwise. To ensure that oblivious rules avoid leaking information, they require that any subexpressions have been fully evaluated before an oblivious expression is reduced. As an example, S-Ctx must be used to reduce the type and payload of an oblivious injection \( \text{in} \) to values before the S-OInj rule can be applied to obtain the oblivious value. The other oblivious rules (e.g. S-Sec and S-Mux) are similar.

The most interesting evaluation rule is S-OCase, which also ensures that an adversary cannot infer anything about the oblivious value being eliminated. In contrast to other oblivious elimination rules like S-Mux, each branch binds the value stored in the sum to its pattern variables. This begs the question of how to instantiate this variable when evaluating the "wrong" branch. Since this branch is eventually discarded when the resulting \( \text{mux} \) is evaluated, we opt to simply instantiate this variable with an arbitrary payload of the right type. This value is synthesized using the auxiliary
relation, \( \tilde{\nu} \leftarrow \tilde{\omega} \), which is also shown in Figure 10. Equipped with this relation, S-OCase reduces to a \( \text{mux} \) expression. As an example, the expression
\[
\text{case } [\text{inr} (<\tilde{\mathbb{B}} \times \tilde{\mathbb{B}}) \Rightarrow \tilde{\mathbb{B} > } [\text{false}]] \text{ of } x_1 \Rightarrow \pi_2 x_1 \mid x_2 \Rightarrow x_2
\]
can step to \( \text{mux } [\text{false}] (\pi_2 ([\text{false}], [\text{true}])) [\text{false}] \). \( x_1 \) is instantiated by ([false], [true]), an arbitrary value of the left injection type.

### 3.3 Type System

The type system of \( \lambda_{OADT} \) ensures that well-typed programs are secure, in that adversaries cannot glean information about private data by observing public information. To guarantee this, kinds in \( \lambda_{OADT} \) are augmented with a security label which constrains how information flows through a program:

\[
\kappa \equiv \\
\quad | \kappa^\text{A} \quad \text{Any} \\
\quad | \kappa^\text{P} \quad \text{Public} \\
\quad | \kappa^\text{O} \quad \text{Oblivious} \\
\quad | \kappa^\text{M} \quad \text{Mixed}
\]

Types that can be treated as either public or oblivious are labeled with \( \kappa^\text{A} \). In practice, this is almost always the unit type, but it includes other singleton types, e.g., \( 1 \times 1 \). Types which are entirely public or entirely private have the labels \( \kappa^\text{P} \) and \( \kappa^\text{O} \), respectively. Finally, types with a mixture of public and private data, e.g., \( \mathbb{B} \times \mathbb{B} \), are labeled with \( \kappa^\text{M} \). This label is also used to classify function types, which we will discuss in more detail shortly. Kinds form a secure join semi-lattice, as shown in Figure 11, with \( \kappa^\text{M} \) being the most restrictive label. Unlike most secure type systems where types with a public label can be promoted to their secure counterparts, in \( \lambda_{OADT} \) public and oblivious labels are not compatible. We elide the security label of a kind when it is not relevant, e.g. \( \Gamma \vdash \tau = \ast \).

Programs in \( \lambda_{OADT} \) are typed using a pair of typing and kinding judgments; we denote these as \( \Sigma \); \( \Gamma \vdash e : \tau \) and \( \Sigma \); \( \Gamma \vdash \tau = \kappa \), respectively. Figure 12 and Figure 13 give the typing and kinding rules for \( \lambda_{OADT} \). We elide \( \Sigma \) from these definitions, as they both assume a fixed global context. For brevity, we omit some side conditions about kinding from the typing rules; these can be found in the Coq development.

The kinding rules for \( \lambda_{OADT} \) are shown in Figure 12. The rules for base types are straightforward. As previously mentioned, function types are assigned a mixed label. The reasons for this are two-fold: firstly, \( \lambda_{OADT} \) does not support oblivious function values. Secondly, this prevents function values from being used as the public view of oblivious types, making it easier for users to be sure oblivious types terminate. The subsumption rule K-Sub allows kinds to be converted to a more restricted label. This rule can be used with K-Prod to label a product type with the join of the labels of its components. K-Sum is similar, but it also includes the public label in the join, as the tag of a public sum is practically public. For example, \( 1 + 1 \), which is equivalent to \( \mathbb{B} \times \mathbb{B} \), should be kinded \( \kappa^\text{P} \) instead of \( \kappa^\text{A} \). Similarly, \( \mathbb{B} + \mathbb{B} \) has to be kinded \( \kappa^\text{M} \), the join of \( \kappa^\text{P} \) and \( \kappa^\text{O} \), as using it in an oblivious context risks leaking the tag. For similar reasons, K-OSum requires the components of oblivious sums to also be oblivious. K-OADT requires the argument of an oblivious type to be well-typed according to its definition in the global context. It does not need to check the index is public, as it is done when typing the global context. K-If and K-Case are the key components for large elimination. They both require the discriminee to be well-typed and the returned types to be obliviously kinded. The K-Case rule is rather permissive in that it does not require the type of discriminee \( e_0 \) to be completely publicly typed. While it is unclear when a programmer would ever...
which allows, for example, reduction under binders and congruence rules. Two terms are then said
with the caveat that they place more constraints on their subterms:

\[ \tau \to_{T-\text{If}} \]

Parallel reduction also plays an important role in the metatheory of \( \lambda_{OADT} \), particularly in the proof
of obliviousness (Theorem 3.7).

A subset of the parallel reduction rules are shown in Figure 14; the remaining rules can be found
in our Coq development; despite their importance in the metatheory of \( \lambda_{OADT} \), the parallel reduction
rules are straightforward. As the figure shows, the rules are essentially more permissive versions
of their counterparts in the step relation from Figure 10. As an example, the parallel reduction rule

\[ \Gamma \vdash \tau : \kappa \]

\[ \text{K-ADT} \]
\[ \text{K-Unit} \]
\[ \text{K-Bool} \]
\[ \text{K-Sub} \]
\[ \text{K-Case} \]
\[ \text{K-Prod} \]
\[ \text{K-Sum} \]
\[ \text{K-OSum} \]
\[ \text{K-OADT} \]
\[ \text{K-If} \]
\[ \text{Fig. 12. Kinding of } \lambda_{OADT} \]

\[ \text{Fig. 12. Kinding of } \lambda_{OADT} \]

actually use a type-level discriminate with oblivious components, it does not leak any information
either.

The typing rules for public constructs are largely standard. Since \( \lambda_{OADT} \) is dependently typed,
T-If and T-Case rely on an implicit motive that is specialized when typing branches\(^2\). This motive,
\((\tau)\), has a special free variable (\(x\)) which stands in for the term being eliminated. The type used
for the then branch in T-If \((\text{true}/x)\) concretizes the occurrences of this variable with \text{true}, for
example. The typing rules for oblivious constructs are largely similar to their public counterparts,
with the caveat that they place more constraints on their subterms: T-0lnj requires the type of its
payload to have an oblivious kind, for example. In addition to requiring that their branches have
oblivious kind, the typing rules for oblivious eliminators (T-Mux and T-0Case) are required to
return non-dependent types, in order to avoid leaking information about the discriminates via their
types. T-BoxedLit and T-BoxedNj type oblivious values, with the latter simply outsourcing it to
the relation used in S-0Case.

The final typing rule, T-Conv, allows any well-typed term to be typed using an equivalent type,
denoted \( \Sigma \vdash \tau \equiv \tau' \). This equivalence is defined directly in terms of a parallel reduction relation, \( \Sigma \vdash \)
e \(\Rightarrow e'\), or simply \( \vdash e \Rightarrow e' \). Parallel reduction is a more liberal version of our call-by-value semantics
which allows, for example, reduction under binders and congruence rules. Two terms are then said
to be equivalent when they can parallel reduce to the same term in zero or more steps:

\[ \Sigma \vdash \tau \equiv \tau' \equiv \tau'' \]. \( \Sigma \vdash \tau \Rightarrow^{*} \tau' \land \Sigma \vdash \tau' \Rightarrow^{*} \tau'' \)

Parallel reduction also plays an important role in the metatheory of \( \lambda_{OADT} \), particularly in the proof
of obliviousness (Theorem 3.7).

\[ \text{This strategy is in line with other dependently typed languages (e.g. Coq), which try to infer a motives when none is}
\]
for \texttt{mux}, \texttt{R-MUX}, does not reduce its branches to values, but immediately takes the corresponding branch, just like \texttt{S-If}. While this rule does leak information whenever the condition of the \texttt{mux} can be reduced to an oblivious value, this situation cannot occur when type-checking the global context, which do not contain oblivious values.

To type a \texttt{\lambdaOADT} program, we also check the definitions in the global context using the rules in Figure 15. \texttt{DT-FUN} is straightforward: the type ascription needs to be well-kindied and the definition needs to be well-typed using an empty typing context. A definition may recursively refer to the name being defined \texttt{x}, which is included in \texttt{\Sigma}. \texttt{DT-ADT}, the typing rule for public ADTs, simply requires the type to be completely public. The typing rule for oblivious ADTs, \texttt{DT-OADT}, requires that its index be completely public, as it is used as the public view. In contrast, the rest of the definition has to have an oblivious kind, under a context that includes the index \texttt{x}.
The proof of Lemma 3.3 depends on two additional lemmas. The first is a regularity lemma needed for the kinding constraints used by several typing rules.

The proof of progress proceeds by mutual induction on typing and kinding derivation. The more general relation.

Fig. 15. Type system of $\lambda_{OADT}$- Global Definition Typing

3.4 Type Safety and Obliviousness

This section presents sketches of the key metatheory proofs for $\lambda_{OADT}$’s type system. All the theorems in this section assume a well-typed global context. Firstly, $\lambda_{OADT}$ enjoys the standard progress and preservation theorems:

**Theorem 3.1 (Progress).** If $\Gamma \vdash e : \tau$, then either $e \rightarrow e'$ for some $e'$, or $e$ is a value.

If $\Gamma \vdash \tau = \kappa^0$, then either $\tau \rightarrow \tau'$ for some $\tau'$, or $\tau$ is an oblivious type value.

The proof of progress proceeds by mutual induction on typing and kinding derivation. The S-OCASE case relies on the fact that every oblivious type value is inhabited, in order to find the oblivious value needed to reduce the “wrong” branch.

The preservation theorem also consists of two parts.

**Theorem 3.2 (Preservation).** If $\Gamma \vdash e : \tau$, and $e \rightarrow e'$, then $\Gamma \vdash e' : \tau$.

If $\Gamma \vdash \tau = \kappa$ and $\tau \rightarrow \tau'$, then $\Gamma \vdash \tau' = \kappa$.

The induction hypothesis for a direct proof of preservation is too weak to prove the T-If and T-CASE cases. Instead, we show that the step relation refines parallel reduction and then prove preservation for the more general relation.

**Lemma 3.3 (Preservation for parallel reduction).** If $\Gamma \vdash e : \tau$, and $e \equiv e'$, then $\Gamma \vdash e' : \tau$.

If $\Gamma \vdash \tau = \kappa$ and $\tau \Rightarrow \tau'$, then $\Gamma \vdash \tau' = \kappa$.

The proof of Lemma 3.3 depends on two additional lemmas. The first is a regularity lemma needed for the kinding constraints used by several typing rules.
Lemma 3.4 (Regularity). If \( \Gamma \vdash e : \tau \) then \( \Gamma \vdash \tau \equiv \kappa \) for some \( \kappa \).

The second is that parallel reduction is confluent.

Lemma 3.5 (Confluence of parallel reduction). If \( e \Rightarrow^{*} e_1 \) and \( e \Rightarrow^{*} e_2 \), then there exists \( e' \) such that \( e_1 \Rightarrow^{*} e' \) and \( e_2 \Rightarrow^{*} e' \).

Interestingly, the regular call-by-value semantics of \( \lambda_{OADT} \) are not confluent, thanks to a combination of the (limited) nondeterminism in S-OCASE and nontermination. Observe that S-OCASE can be applied with different choices of the arbitrary oblivious values. This is not a problem if the oblivious case expression terminates because the "wrong" branch will eventually be discarded. However, it is possible that the \( S\text{-Mux} \) expression it steps to loops forever, such that the "wrong" branch is never discarded. Thankfully, the R-Mux rule is more liberal than S-Mux, ensuring that parallel reduction is confluent. Whenever \( \text{case} \) parallel reduces to a \( \text{mux} \) expression, however, R-Mux will immediately discard the "wrong" branch, forcing both choices to converge within one step.

3.4.1 Obliviousness. Adversaries should not be able to infer any information about the private information (i.e. oblivious values) of well-typed \( \lambda_{OADT} \) programs by observing the whole execution of a \( \lambda_{OADT} \) program. To prove this, we first formalize a notion of indistinguishability for \( \lambda_{OADT} \) expressions:

Definition 3.6 (Indistinguishability). We say two expressions are indistinguishable, denoted by \( e \approx e' \), if
1. they are both oblivious Boolean values: \([b] \approx [b'] \), or
2. they are both oblivious injections with the same type: \([\text{in}_{\tau} \llcorner \omega \rangle \ \nu] \approx [\text{in}_{\tau} \llcorner \omega \rangle \ \nu'] \), or
3. they are the same expression with indistinguishable sub-expressions.

Intuitively, two expressions are indistinguishable if they only differ in their oblivious values.

Equipped with this relation, we can now formally state the obliviousness theorem for \( \lambda_{OADT} \):

Theorem 3.7 (Obliviousness). If \( e_1 \approx e_2 \) and \( \cdot \vdash e_1 : \tau_1 \) and \( \cdot \vdash e_2 : \tau_2 \), then
1. \( e_1 \rightarrow^{n} e'_1 \) if and only if \( e_2 \rightarrow^{n} e'_2 \) for some \( e'_2 \)
2. \( e_1 \rightarrow^{n} e'_1 \) and \( e_2 \rightarrow^{n} e'_2 \), then \( e'_1 \approx e'_2 \)

We write \( e \rightarrow^{n} e' \) to mean \( e \) reduces to \( e' \) in exactly \( n \) steps. The first piece of this theorem is a generalization of progress, and ensures that information is not leaked via a termination channel.

The second piece says that for any two indistinguishable programs, an observer cannot learn anything about their oblivious values by examining the states they can step to. Taken together, these two properties ensure that an observer cannot learn anything about the private values in a well-typed \( \lambda_{OADT} \) program, even given the entire execution trace of that program. If we treat the observable parts of the intermediate execution states as a public channel, obliviousness provides a sort of noninterference property [Goguen and Meseguer 1982; Sabelfeld and Myers 2003], in that different private (i.e. high-security) inputs do not leak any information via this public channel.

The proof of Theorem 3.7 is by induction on the derivation of \( e_1 \rightarrow^{n} e'_1 \). The first part of the proof of obliviousness is a direct consequence of progress and the fact that well-typed values are only indistinguishable from other values. The second part is more involved, and requires the following two key lemmas to prove the S-Mux case:

Lemma 3.8. If \( \Gamma \vdash \nu : \tau \) and \( \Gamma \vdash \nu : \tau' \), and \( \Gamma \vdash \tau \equiv \kappa \), then \( \nu \equiv \nu' \).

Lemma 3.9. If \( \nu \equiv \nu', \Gamma \vdash \nu : \tau, \Gamma \vdash \nu' : \tau', \) and \( \Gamma \vdash \tau \equiv \kappa \), then \( \tau \equiv \tau' \).
Lemma 3.8 states that all values of the same oblivious type are indistinguishable, and Lemma 3.9 ensures that two indistinguishable, obliviously-typed values have the same type up to type equivalence. The proofs of both lemmas proceed by induction on the typing derivation. Most of the proofs are straightforward, except for the case of T-Conv in Lemma 3.9. Since applying the induction hypothesis requires that \( \tau' \) also be oblivious, we need to show that two equivalent, well-kinded types simultaneously have oblivious kinds:

**Lemma 3.10.** If \( \tau \equiv \tau' \), \( \Gamma \vdash \tau :: \ast_0 \), and \( \Gamma \vdash \tau' :: \ast_0 \), then \( \Gamma \vdash \tau' :: \ast_0 \).

### 4 \( \lambda_{OADT}^\bullet \), FORMALLY

This section formalizes an \( \lambda_{OADT}^\bullet \), an extension to \( \lambda_{OADT} \) that permits implementations in the vein of Figure 6.

#### 4.1 Syntax

The extended syntax of \( \lambda_{OADT}^\bullet \) is shown in Figure 16. These extensions permit \( \lambda_{OADT}^\bullet \) expressions that potentially leak information locally, as long as they can eventually be repaired by the surrounding context. The new \( \text{if} \) operation is similar to \( \text{mux} \), but its branches are permitted to be non-oblivious, causing a potential leak if \( \text{if} \) is evaluated naively. The new \( \text{tape} \) annotation acts as a boundary for potential leaks, and is used to ensure that they never occur during execution, as Section 4.2 will discuss in more detail. Finally, \( \lambda_{OADT}^\bullet \) updates the syntax for anonymous functions, function types and function definitions with a leakage label. A leakage label is either \( \top \) or \( \bot \), and signals either the presence or the absence of a potential leak, respectively.

#### 4.2 Semantics

The semantics of \( \lambda_{OADT}^\bullet \) are an extension of the semantics of \( \lambda_{OADT} \). Figure 17 shows the new and updated rules; the rest are identical to the rules in Figure 10. This semantics introduces a new syntactic class of weak values, which are used to ensure that \( \text{if} \) does not leak information when evaluated. Weak values simply extend the values in \( \lambda_{OADT} \) with \( \text{if} \): a \( \text{if} \) is a weak value if all its subexpressions are weak values. All references to \( v \) in the reduction rules (including those not shown in Figure 17) now refer to weak values unless explicitly identified as a value. The semantics also extend evaluation contexts to handle \( \text{if} \) and \( \text{tape} \) expressions.

The S-\( \text{OiF} \) rule captures the key idea of distributing surrounding context into the branches of \( \text{if} \). Like S-MUX, this rule requires its branches to first be evaluated to weak values using S-Ctx. Note that not all contexts need to be distributed into these branches in order to make progress—pushing fold into \( \text{if} \) in the expression \( \text{fold<tree>} \ (\text{if} \ {\text{[true]}} \ldots) \) does not gain us anything, for example, since the expression is already a weak value. Figure 17 defines the leaky contexts \( \mathcal{E} \) that can be distributed through \( \text{if} \). For simplicity, we adopt a minimal set of leaky contexts, though allowing...
more contexts is a potential avenue for optimizing executions. This does not limit the expressivity of \( \lambda_{OADT^\bullet} \), for similar reasons to the fold example from above. The semantics of \( \text{case} \) are also updated to allow potential leaks, with S-OCASE now evaluating to \( \text{if} \) instead of \( \text{mux} \).

The last three rules in Figure 17 show how to evaluate \( \text{tape} \) annotations. The key idea is to use \( \text{tape} \) as a signal that the context surrounding an \( \text{if} \) expression has been sufficiently distributed to prevent leaks. Mechanically, whenever a \( \text{tape} \) annotation is applied to an \( \text{if} \) expression whose branches are weak values, it is safe to reduce expression to a secure \( \text{mux} \) using S-TapeOIf. As an example, consider the following expression:

\[
\text{tape} \left( \text{if} \left( \text{true} \right) \text{then} \text{false} \text{else} \text{true} \right) \rightarrow \text{tape} \left( \text{if} \left( \text{true} \right) \text{then} \left( \text{tape} \left( \text{false} \right) \right) \text{else} \left( \text{tape} \left( \text{true} \right) \right) \right) \rightarrow^* \left( \text{tape} \left[ \text{false} \right] \right) \rightarrow^* \left[ \text{false} \right]
\]

After applying S-OIf to distribute the surrounding Boolean section \( s_\text{B} \), the \( \text{if} \) expression is now annotated with \( \text{tape} \), and S-TapeOIf can be applied. The \( \text{tape} \) annotations are pushed inside the branches of \( \text{mux} \) to ensure any \( \text{if} \) expressions they may contain are also repaired. The final two rules ensure \( \text{tape} \) annotations are eventually dropped from oblivious values. An oblivious (non-pair) value annotated with \( \text{tape} \) cannot leak any information, and S-TapeOVal can be applied to remove the extraneous \( \text{tape} \) annotations to be distributed into the components of an oblivious pair, in order to eventually repair any \( \text{if} \) expressions they may contain.
purposes. This coarse-grained analysis is strong enough for our

\[ \Gamma \vdash e : \tau \]

4.3 Type System

The typing judgment of \( \lambda_{OADT} \) now includes a leakage label for the typed expression: \( \Gamma \vdash e : \tau \), as do entries in typing contexts \( \Gamma \). Figure 18 shows a subset of the typing rules of \( \lambda_{OADT} \); the omitted rules are copies of those from \( \lambda_{OADT} \) with straightforward leakage labels annotations. As mentioned in Section 4.1, leakage labels signal whether an expression might contain a potential leak. The reason for these labels is similar to the security labels found in other security-type systems [Sabelfeld and Myers 2003; Zdancewic 2002], where type-based information flow control is used to enforce noninterference between high- and low-security information. In \( \lambda_{OADT} \), expressions with \( \tau \) labels should not influence expressions with \( \bot \) labels. In order to minimize the extension to \( \lambda_{OADT} \), we do not annotate every type with a leakage label, opting to only annotate top-level definitions and function parameters with leakage labels. This coarse-grained analysis is strong enough for our purposes.

The leakage label of base types is always \( \bot \), e.g., in T-UNIT. Leakage labels for local or global variables is taken directly from the context, e.g., T-VAR. For most public constructs, e.g., T-PAIR and T-PROJ, the label is the join (\( \sqcup \)) of the labels of all sub-expressions, where \( \bot \sqcup \bot = \bot \) and \( \top \) otherwise. T-PROJ shows why leakage is an overapproximation, as we cannot always tell which component of a pair labeled with \( \top \) is the source of the potential leak. In T-ABS, both the type and label of a parameter are added to the typing context when typing the function body. T-APP requires a function to be applied to an argument whose label matches that of its parameter. Applying a function with a potentially leaky parameter to a non-leaky argument can be typed by first using the T-CONV rule, which allows the label of an expression to be downgraded. As an example, these rules ensure both \((\lambda x: A \Rightarrow B) \text{true} \) (if \([\text{true}] \text{then true else false}\)) and \((\lambda x: A \Rightarrow B) \text{true} \) are well-typed expressions.

\( \lambda_{OADT} \) has dependent and nondependent versions of the typing rule for if. In the dependent version, T-IF, the discriminee is not allowed to contain a potential leak, as it may appear in the type. In the nondependent version T-IFNoDep, there is no such restriction, but the type is not allowed to depend on the discriminee. The typing rules for case, T-CASE and T-CASENoDep, are similar.

The remaining typing rules deal with expressions that either repair or introduce potential leaks. An expression annotated with tape is always assigned the \( \bot \) label, as long as that expression has an oblivious type. This is in line with the semantics of tape: when applied to an oblivious expression, it eventually evaluates to an oblivious value or a weak value (\( \epsilon \)). The former is already safe, and the latter can be repaired by S-TAPEOIf. The \( \bot \) label in the rule captures the idea that tape safely repairs a local leak, such that the surrounding computation can treat it as non-leaky. The rules for \( \epsilon \text{if} \) and \( \epsilon \text{case} \) reflect the fact that they are sources of potential leaks, as both expressions are labelled with \( \top \). Both rules require their discriminees to be free of potential leaks, but this does not affect expressiveness, since their discriminees can always be wrapped with tape. T-OINJ and T-MUX feature similar requirements.

Figure 19 shows the updated kinding rules for \( \lambda_{OADT} \); the other kinding rules are identical to those in Figure 12. The updated rules require types to only depend on terms that do not contain potential leaks, i.e. those assigned the \( \bot \) label. To see why, consider the following ill-kindred type:

\[
\text{if} \ \epsilon \text{if} \ [\text{true}] \text{then true else false} \ \text{then} \ 1 \ \text{else} \ \hat{B}
\]

After distributing the surrounding \( \epsilon \text{if} \), this reduces to \( \epsilon \text{if} \ [\text{true}] \text{then true else false} \). Similar expressions at the term level can be repaired by, e.g., distributing \( s_B \) through the branches to secure the result of the \( \epsilon \text{if} \). At the type level we have no such recourse, however: since types are always public, there is no corresponding way to repair this type by securing its branches.

Figure 20 shows a subset of the updated and new parallel reduction rules. Again, the rules for oblivious constructs are similar to the corresponding step rules. In R-OIf, we write \( \hat{E} \Rightarrow \hat{E}' \) to mean...
all the subexpressions in the leaky context take a parallel reduction step. R-IfLeak is required for confluence, similar to R-Mux. We say a λOADT+ program is well-typed if the global context is well-typed (the updated typing rules for the global context are trivial) and the expression is well-typed with ⊥ label. The latter restriction ensures that all potential leaks in a λOADT+ program are eventually repaired.

4.4 Type Safety and Obliviousness

The guarantees of the type system of λOADT+ are quite similar to those of λOADT, although they have been adapted slightly to account for leakage labels. The statement of progress for λOADT+, for example, is limited to expressions without potential leaks:

\[ \text{Theorem 4.1 (Progress). } \text{If } \vdash e : \tau, \text{ then either } e \rightarrow e' \text{ for some } e', \text{ or } e \text{ is a value.} \]

\[ \text{If } \vdash \tau : \ast^O, \text{ then either } \tau \rightarrow \tau' \text{ for some } \tau', \text{ or } \tau \text{ is an oblivious type value.} \]

This updated statement reflects the fact that leaky expressions only reduce to weak values. The proof of this theorem is a consequence of a stronger lemma which also accounts for potentially leaky expressions:

\[ \text{Lemma 4.2. } \text{If } \vdash e : \tau, \text{ then either } e \rightarrow e' \text{ for some } e', \text{ or } e \text{ is a weak value.} \]

The proof of this stronger lemma proceeds similarly to the proof of progress for λOADT, with the canonical form lemmas extended to weak values. One technicality needed by this proof is a notion of weak oblivious value, which extends oblivious values to include \( \tilde{\text{if}} \) expressions. For the T-Tape case, we have to show that a weak value with an oblivious type is also a weak oblivious value, as \( \text{tape} \) can only be reduced when it is applied to weak oblivious values. This extra lemma requires an updated version of Lemma 3.10, so the proof of progress for λOADT+ now depends on type preservation for parallel reduction. With this lemma in hand, the progress theorem immediately follows from the fact that a weak value is a value if it labeled with ⊥.

The statements of preservation and obliviousness must also be updated to deal with leakage labels, but are otherwise identical.
This section considers how additional base types might be added to the core calculus of \( \lambda_{OADT} \), using fixed-width integers as an example. Figure 21 shows a subset of the syntax, semantics and typing rules needed for this new primitive type. The extended language includes public and oblivious versions of integer types, literals, and operators. For simplicity, we only consider a comparison operation, but additional operators could be added in a similar manner. In order to move between the public and oblivious types, section (\( SI-Sec \)) and retraction (\( SI-Ret \)) operations for integers are also added; both have similar semantics to their Boolean counterparts. \( r_B \) always introduces a potential leak, and \( r_Z \) is considered weak value.

When defining the semantics of potentially leaky expressions like \( \leq \), it is important that the semantics does not leak information via the execution trace. When comparing oblivious values with \( \leq \), for example, SI-RetLE1 combines \( \leq_Z \) and \( r_B \) to first securely compare the operands before retracting the resulting oblivious Boolean. SI-RetLE2 and SI-RetLE3 are similar, but they apply to cases when one of the operands is not a retraction of an oblivious value by lifting it to oblivious values first. The semantics of other operators can be defined through similar uses of section and retraction functions. As an example, integer addition returns an integer instead of Boolean, so we apply \( r_Z \) to the result of oblivious addition. If a leaky integer expression is used in a well-typed

\[ e, \tau \coloneqq \begin{cases} \ldots & \\
Z \mid \bar{Z} & \text{Primitive integer types} \\
i \mid [i] & \text{(Runtime) integer literals} \\
e \leq e \mid e \preceq e & \text{Integer operators} \\
s_Z \mid r_Z e & \text{Integer section and retraction} 
\end{cases} \]

\[ \Gamma \vdash e : \tau \]

\begin{align*}
\text{(a) Syntax} & & \text{(b) Typing} \\
\Gamma \vdash \tau = \kappa & & e \rightarrow e' \\
\text{SI-Sec} & \quad \text{SI-SecRet} & \quad \text{SI-RetLE} \\
s_Z i \rightarrow [i] & \quad s_Z (r_Z [i]) \rightarrow [i] & \quad r_Z [i_1 \leq i_2] \rightarrow r_B ([i_1] \preceq [i_2]) \\
\text{SI-RetLE}_2 & & r_Z [i_1 \leq i_2] \rightarrow r_B ([i_1] \preceq [i_2]) \\
\text{SI-RetLE}_3 & & i_1 \leq r_Z [i_2] \rightarrow r_B (s_Z i_1 \preceq [i_2]) \\
\end{align*}

\begin{align*}
\text{(c) Semantics} \\
\text{Theorem 4.3 (Preservation).} & \quad \text{If } \Gamma \vdash e : \tau, \text{ and } e \rightarrow e', \text{ then } \Gamma \vdash e' : \tau. \\
& \quad \text{If } \Gamma \vdash \tau = \kappa \text{ and } \tau \rightarrow \tau', \text{ then } \Gamma \vdash \tau' = \kappa. \\
\text{Theorem 4.4 (Obliviousness).} & \quad \text{If } e_1 \approx e_2 \text{ and } \vdash e_1 : \tau_1 \text{ and } \vdash e_2 : \tau_2, \text{ then} \\
& \quad (1) e_1 \rightarrow^n e_1' \text{ if and only if } e_2 \rightarrow^n e_2' \text{ for some } e_2' \\
& \quad (2) \text{if } e_1 \rightarrow^n e_1' \text{ and } e_2 \rightarrow^n e_2', \text{ then } e_1' \approx e_2'. \\
\end{align*}

Proofs of both theorems follow the same structure as their counterparts in \( \lambda_{OADT} \), although many of the lemmas used in the proof of obliviousness now use weak values instead of values.

4.5 Extending \( \lambda_{OADT} \)

Although \( \lambda_{OADT} \) does not include Boolean retraction \( r_B \) as a primitive, it is easily defined in terms of \( \overline{\text{if}} \):

\[ r_B e \triangleq \overline{\text{if } e \text{ then true else false}}. \]
context, then $r_Z$ will eventually meet $s_Z$ and they can be cancelled out via SI-SecRet. Updated versions of evaluation contexts, leaking contexts and the other reduction rules are omitted, as they are straightforward extensions of their counterparts in $\lambda_{OADT^+}$. The extended typing and kinding rules are also straightforward, and are similar to those for the primitive types in $\lambda_{OADT^+}$. Figure 21 gives the rules for integer retraction (TI-RET) and oblivious less-than (TI-OLE) as examples.

### 4.6 $\lambda_{OADT^+}$ in action

To demonstrate the expressiveness of $\lambda_{OADT^+}$, we have written some example oblivious functions and oblivious types with different public views. We have directly encoded these in our Coq development, as well as some accompanying typing and evaluation derivations. All of the examples described in this section are available in the supplementary material.

We have encoded the following OADTs for lists and trees. Each oblivious type consists of its type definition, a section function and a retraction function.

- List with the upper bound of its length.
- Tree with the upper bound of its depth.
- Tree with the upper bound of its spine.
- Tree with the upper bound of the number of its vertices (including leaves and nodes).

The second and third of these examples were presented in Section 2. The oblivious tree with the upper bound of its total vertices is the most complicated: while its type definition is effectively an oblivious list, its section and retraction functions correspond to flattening a tree and rebuilding a tree from a list.

In addition to the lookup function from Section 2, we have also written a tree insertion function as a demonstration of how oblivious ADTs are constructed. A more interesting example is a standard map function over oblivious trees, which shows that higher-order functions can be naturally written in $\lambda_{OADT^+}$. The following code snippet for an oblivious map function follows the recipe in Figure 6. Label annotations are omitted for brevity, and we use a Boolean payload for simplicity.

```coq
def map (f : B → B) (k : N) (t : tree k) : tree k :=
S_tree (map f (r_tree k t)) k
```

The function argument of $\map$ takes a public Boolean to public Boolean, but $\map$ could be adapted to accept a function from oblivious Boolean to oblivious Boolean by composing Boolean section and retraction to appropriately "transport" the function argument. The $\map$ function could also be adapted any oblivious tree definition by simply replacing $\tree$, $\s_tree$, and $\r_tree$.

### 5 RELATED WORK

Secure computation was first formally introduced by Yao [1982] alongside his proposed solution, Garbled Circuits. In secure computation, an untrusted party may observe the whole execution of the secure program, or infer some private information from other side-channels. Enabling secure computations that use algebraic data types that also hide their structures is a key motivation of this work. Secure computation techniques can be broadly divided into those using multiparty computation and those relying on outsourced computation [Evans et al. 2018; Hazay and Lindell 2010]. Those in the former category typically use protocols based on either Garbled Circuits or secret-sharing schemes [Beimel 2011; Goldreich et al. 1987; Maurer 2006]. In the realm of outsourced computation, solutions are typically based on fully homomorphic encryption [Acar et al. 2018; Gentry 2009], but can also be supported by virtualization [Barthe et al. 2014, 2019a] or secure processors [Hoekstra 2015]. These protocols can be used to implement the semantics of $\lambda_{OADT}$ and $\lambda_{OADT^+}$.
Many high-level programming languages have been proposed that support some form of secure computation [Hastings et al. 2019]. Their goals are similar to ours in that they provide high-level language support for writing secure programs. However, most do not support (recursive) data structures at all, or assume the structural information is always public. Obliv-C [Zahur and Evans 2015] is a C-like oblivious language. Algebraic data types can be encoded with the C-style `struct` keyword with pointers. Since their oblivious types are restricted to base C types, however, the structure of the defined ADT is public. It would be possible to implement oblivious ADT in Obliv-C by manually padding and using the data types according to their public views. The language provides a `~obl1v` keyword that can be used to dynamically track the maximum bound of a data type, at the cost of some additional user effort. Moreover, if the programmers decide to use a different public view, they have to fix every places where this data type is used. ObliVM [Liu et al. 2015] is a Java-like language which also has a `struct` keyword to define data types, but only supports public structures, much like Obliv-C. Wysteria and Wys∗ [Rastogi et al. 2014, 2019] are functional languages that focus on mixed-mode computation. While they do not support recursive data types, both languages include simple polynomial types and primitive arrays. In contrast, our languages does not consider mixed-mode computation.

Constant-time languages protect programs from inadvertently leaking private information through timing channels by providing atomic constructs and carefully tracking information control. This is also a goal of our system, and our solution to this problem is similar. The first formal study of constant-time algorithms was in the context of cache-based attacks [Barthe et al. 2014]. Barthe et al. [2019b] extended the formally verified CompCert compiler [Leroy 2009] to ensure constant time execution. Though we do not have a compiler for λOADT or λOADT✚, our obliviousness theorem does provide a formal guarantee of a constant-time property. FaCT [Cauligi et al. 2019] is a high-level language for writing constant-time computation using (non-recursive) data types. One of its unique features is a front-end compiler to transform a well-typed (but potentially not constant-time) FaCT program to a constant-time FaCT program. In λOADT✚, the programmers can simply encode programs in the conventional fragment and then lift them to oblivious programs (that are constant-time) by composing section and retraction functions.

λobliv [Darais et al. 2020] is a functional programming language for oblivious computation that focuses on probabilistic programs, making it suitable for implementing some oblivious cryptographic algorithms, such as ORAM, although it does not include algebraic data types. In contrast, our work does not consider probabilistic programs, though it could be an interesting future direction. Our two approaches share similar threat models and guarantees of obliviousness.

Our approach of type-based information flow control to enforce obliviousness, a form of noninterference, follows a body of work in security-type systems [Sabelfeld and Myers 2003; Zdancewic 2002]. To the best of our knowledge, our system is the first to combine a dependent type system with large elimination and a security-type system. Our notion of retraction bears some resemblance to delimited information release [Sabelfeld and Myers 2004]. In a system with delimited information release, the programmers may choose to reveal some private information, similar to retraction functions in λOADT✚. However, our semantics guarantees retraction never releases any private information. Another difference with a standard security-type system is that we use explicit coercion via section functions instead of implicit subtyping to convert public types to secure types. On the one hand, our typing rules and semantics for oblivious types and non-oblivious types are quite different. On the other hand, implicit subtyping does not make sense in the case of ADTs. To convert a public ADT to an oblivious one, we not only need to know how the oblivious ADT is represented, but also to infer the public views.

Our oblivious types can also be viewed as a kind of refinement types [Kawaguchi et al. 2009; Rondon et al. 2008; Xi and Pfenning 1999]: the oblivious tree in our running example can be
understood as trees with a maximum depth stipulated by the type index, for example. However, this declarative specification does not explain how to represent such an oblivious tree. Nonetheless, this view of subset types suggests a future direction of integrating refinement typing into our system to ensure the correct use of public indices. Dependent type systems with large elimination can be found in many theorem provers, such as Coq [The Coq Development Team 2021] and Agda [Norell [n.d.]]. These languages are designed more towards theorem proving and thus only admit total functions, while our languages allow general recursion and hence nontermination. A notable dependently typed languages with nontermination is Zombie [Sjöberg 2015; Sjöberg et al. 2012; Sjöberg and Weirich 2015], though our goals are drastically different.

In our mechanized formalization, the correctness and security guarantees provided by the underlying cryptographic primitives are baked into our semantics and notion of indistinguishability. There is a body of work about formally verified cryptography [Abate et al. 2021; Barthe et al. 2011, 2009], which could be integrated into our work in the future to provide a stronger formal guarantee. Some of these solutions have focused on verifying multiparty computation [Backes et al. 2010; Haagh et al. 2018].

Another popular cryptographic technique for hiding private information of data structures is oblivious RAM [Goldreich 1987; Goldreich and Ostrovsky 1996; Stefanov et al. 2013] (ORAM). ORAM provides primitives to access an encrypted memory buffer without revealing the access pattern, except for the number of accesses. There have been proposals for generically constructing oblivious data structures using ORAM [Wang et al. 2014]. Oblivious data structures constructed this way hide the access patterns of a sequence of data structure operations. This line of work in general does not consider leakage through side-channels. While our solution also naturally hides access patterns, we also assume a much stronger adversary who can observe the whole computation.

6 DISCUSSION AND FUTURE WORK

In contrast to our “dynamic” approach of repairing potential leakage in \( \lambda_{OADT} \), it is also possible to repair programs statically, at least for some simple programs. The key observation is that we can apply fusion techniques [Ohori and Sasano 2007; Wadler 1990] to implementations like the one in Figure 8. By fusing the composition of section, public function and retraction, it is possible to build an implementation from oblivious input to oblivious output, without the intermediate public trees generated by the retraction function. While this method works for some simple examples like an append function for lists, only a limited amount of functions can be fully fused. Nonetheless, we believe this is a direction worth exploring in the future.

One natural next step is the implementation of an algorithmic type checker, although the possibility of nontermination in our languages poses a potential challenge to dependent type checking. One solution is to simply ask the programmers to provide a “fuel” to bound how many steps the type checker can take, similar to Zombie [Sjöberg et al. 2012]. We could also only allow total functions at the type level. As dependent types only occur in section and retraction functions, which have rather specialized forms, we believe a weaker and incomplete type checker would work in practice. Building an implementation of \( \lambda_{OADT} \) is also an important piece of future work, where performance may play a critical role. While oblivious programs are generally quite slow in practice due to timing channel protections, our nonconventional semantics may introduce additional overheads.

Other future directions include the formalization of a general algorithm for synthesizing implementations like Figure 8, synthesizing section and retraction functions, inference of leakage labels, and automatic insertion of \texttt{tape} annotations.
7 CONCLUSION

To our best knowledge, this work is the first programming language that supports hiding the structure of rich recursive data types in secure computations. We have presented $\lambda_{OADT}$, a core calculus for encoding oblivious programs over oblivious algebraic data types. $\lambda_{OADT}$ combines dependent types with large elimination to represent oblivious algebraic data types, and provides a security-type system to ensure that computations reveal no private information over what is provided by the public view of the data. To enable programmers to write a single function and easily build secure programs with different public views, we have also developed $\lambda_{OADT}$. This language is equipped with a novel semantics that repairs potential leaks without compromising the security guarantees of $\lambda_{OADT}$. We have proved, mechanically, that our solution provides a strong and formal security guarantee: an adversary can not infer any private information, even given the entire execution trace of a program.

REFERENCES


Oblivious Algebraic Data Types

arXiv:1202.2923


