Introduction

This is the artifact for the OOPSLA 2024 paper "Taypsi: Static Enforcement of Privacy Policies for Policy-Agnostic Oblivious Computation". The artifact includes the source code of the lifting algorithm (Section 5) and the Taypsi language, a Coq formalization of Taypsi core calculus (Section 3), and all of the benchmarks from the our evaluation (Section 6).

This artifact supports the following claims made in the paper:

- When applied to algorithms involving intermediate values of structured data types, Taypsi lifting algorithm produces programs that perform considerably better than the baseline established in prior work.
- When applied to algorithms that only construct primitive values, Taypsi's lifting procedure produces programs that perform on par with the baseline approach.
- The compilation overhead introduced by Taypsi's lifting algorithm is reasonable.
- The core calculus and proofs of soundness and obliviousness have been mechanized in Coq.

Reproducing the experimental results explains how to reproduce the experiments that support the first three claims, and Coq formalization of the core calculus gives instructions on how to navigate the Coq mechanization.

Hardware Dependencies

The artifact is packaged as docker images for amd64 (x86-64) and arm64 architectures. Thus, any hardware using these two architectures should work, as long as the installed operating system is supported by docker. However, fully reproducing our experimental results requires at least 8 GB of memory. The docker image also requires roughly 14 GB of storage space.

We have tested this artifact on a x86-64 Linux box and an Apple Silicon (M1) Mac.

Getting Started Guide

This artifact is a docker image, which contains:

- This README file, located at `~/README.md`. A rendered version is also available online.
- The docker file used to generate the docker images, located at `~/Dockerfile`.
- The implementation of the Taypsi type checker and compiler, based on Taype (PLDI23), located at `~/taypsi`. (Github repository)
- The implementation of the baseline Taype type checker and compiler used in our evaluation, located at `~/taype-pldi`. This includes the additional benchmarks used in our comparison. (Github repository)
- An extended version of Taype that includes Taypsi's smart array optimization, located at `~/taype-sa`. This version of Taype is used to provide a fairer comparison with Taypsi. (Github repository)
- All examples and experiments from the paper, located at `~/taypsi/examples` (correspondingly `~/taype-pldi/examples` and `~/taype-sa/examples`).
- The Coq formalization of the Taypsi core calculus, based on Oblivious Algebraic Data Types (POPL22), located at `~/taypsi-theories`. (Github repository)
- The source code of drivers implementing the cryptographic primitives and oblivious array, located at `~/taype-drivers`. This implementation includes the smart array optimization, and is used by

Taypsi (`~/taypsi`) and the version of Taype extended with the smart array optimization (`~/taype-sa`). (Github repository)

- The source code of the drivers from Taype (PLDI23), located at `~/taype-drivers-legacy`. This implementation is used by the baseline Taype (`~/taype-pldi`). (Github repository)
- A code-server (VS Code in the browser), so that users can more easily browse the source code (this is not required, of course). We have include the following VS Code extensions:
  - Taype: for reading Taypsi source code. This extension provides basic syntax highlighting for Taypsi and its intermediate language OIL. (The name of this extension reflects that Taypsi is based on and is an extension of Taype)
  - Haskell: for reading source code of the Taypsi type checker and compiler, which is implemented in Haskell.
  - OCaml: for reading source code of the generated OCaml programs, test cases and part of the source code of Taypsi. Since Taypsi programs are compiled to OCaml libraries, our test cases are also written in OCaml, which handle I/O and invoke these libraries. The constraint solver presented in the paper is also implemented in OCaml.
  - VsCoq: for reading Coq formalization.
  - Python: for reading the script that interprets the evaluation results and generates LaTeX tables.

All the implementations in the docker image have been pre-compiled. The clean version of the source code, this README file and the docker file are also available on Zenodo.

To evaluate this artifact, first install docker, and then download one of our docker images from Zenodo, depending on your machine's architecture. We provide images for amd64 (i.e. x86-64) and arm64 (e.g., for Apple Silicon Mac). You need around 14 GB of storage space to load these images, and 8 GB of RAM for the container to run the experiments.

Before executing any docker commands, make sure that the docker daemon is running: if you see `Cannot connect to the Docker daemon` in the output of command `docker version`, then you need to start the daemon first. Check the docker official documentation for instructions according to your operating system and docker version.

Now you can load and run the downloaded docker image. The following commands create an image called `taypsi-image`, and start a container called `taypsi`. We expose the port `8080` for accessing the code-server.

```
# <arch> is amd64 or arm64
mv taypsi-image-<arch>.tar.xz taypsi-image.tar.xz
# This command will take a minute or two
docker load -i taypsi-image.tar.xz
docker run -dt -p 8080:8080 -m 8g --name taypsi taypsi-image
```

The docker container is allocated 8 GB of memory which is the memory cap used in the evaluation section. You could allocate a smaller amount of memory if 8 GB is not possible, but you would not be able to completely reproduce the experimental results (some benchmarks may fail). You need around 2 GB to compile the Coq formalization.

To launch the code-server, run:

```
docker exec -d taypsi code-server
```

Now you can open the URL localhost:8080 (or 127.0.0.1:8080) in a browser to access VS Code. Note that some functionality may not work if you are using private mode or incognito mode. We did not pre-install the Haskell language server or the OCaml language server in the docker image; the next section includes instructions on how to do so.

To access the container shell, run

```
docker exec -it taypsi bash --login
```

Your user name is `reviewer` (without password) and the current directory is `~` (i.e. `/home/reviewer`). In the rest of this document, we assume commands are run inside the container.

To quickly test this artifact, compile the tutorial example and run its test cases. The Taypsi source file of this example `taypsi/examples/tutorial/tutorial.tp` includes comments on how to write Taypsi programs and oblivious types.

```
cd taypsi
cabal run shake -- run/tutorial
```

We will explain what exactly this command is doing in the next section, but you should see the output of the tests, which contains headers like:

```
== Test case 1 (round 1) ==
```

and then a few numbers for the performance statistics.

Step-by-Step Instructions

This section provides details on how the figures (claims) in the paper correspond to the implementation, how to reproduce the experimental results, how to use our tools, and some minor discrepancies between the implementation and the paper's description.

How to read code

As mentioned in the previous section, you can read the source code in the browser with code-server. The docker image also comes with vim, if you prefer reading source code in the console, but we do not have a syntax highlighting extension for vim yet.

You may want to install Haskell and OCaml language servers for richer IDE features such as jump to definition. You can install them by running:

```
# Install Haskell language server
# This step may not be needed, as the Haskell VS Code extension may ask and do this
for you
ghcup install hls

# Install OCaml language server
opam install ocaml-lsp-server
```

Correspondence between paper and artifact

The following table describes how various definitions in the paper correspond to definitions in the artifact.

Note that we still use the name Taype (e.g., in file names and module names) in the Taypsi compiler source code, as Taypsi is an extension of Taype.

In paper	In artifact	Comment
Fig. 1	`list` and `filter` in `taypsi/examples/tutorial/tutorial.tp`	See Note 1
Fig. 2	`~list` and `~list_eq` in `taypsi/examples/tutorial/tutorial.tp`	
Fig. 4	`~list#s`, `~list#r`, `~list#view`, `~list#Nil`, `~list#Cons`, `~list#match`, `~list#join` and `~list#reshape` in `taypsi/examples/tutorial/tutorial.tp`	
Figures and theorems in Section 3	See Coq formalization of the core calculus	
Fig. 14	`liftDefs` in `taypsi/src/Taype/Lift.hs`	See Note 2
Fig. 15	`Ppx` in `taypsi/src/Taype/Syntax.hs` and `elabPpx` in `taypsi/src/Taype/TypeChecker.hs`	Typed macros are called preprocessors (ppx) in source code
Fig. 16	`Constraint` in `taypsi/src/Taype/Lift.hs`	
Fig. 17	`liftExpr` in `taypsi/src/Taype/Lift.hs`	
Compilation and optimizations in Section 6	Source code in `taypsi` and `taype-drivers`	See Note 3
Figures in Section 6	See Reproducing the experimental results	

Notes:

1. The Implementation distinguishes between oblivious product (whose components must be oblivious) and normal product (whose components can be any types), similar to Ye and Delaware (PLDI23). The style in the Taypsi paper is closer to Ye and Delaware (POPL22), which includes only one product former that can connect any types, for presentation purposes.
2. While the entry point of the lifting algorithm is `liftDefs` in `taypsi/src/Taype/Lift.hs`, some subroutines are implemented in other files: the source of our constraint solver can be found in `taypsi/solver/bin/solver.ml` and `taypsi/solver/lib/solver.ml`, and the elaborator of typed macros is the `elabPpx` function in `taypsi/src/Taype/TypeChecker.hs`.
3. The source code of the bidirectional type checker is in `taypsi/src/Taype/TypeChecker.hs`, the lifting procedure is in `taypsi/src/Taype/Lift.hs`, and the translation to OIL is in `taypsi/src/Oil/Translation.hs`. The smart array optimization is defined in `taype-drivers/lib/smart.ml`. The reshape guard optimization is defined at `guardReshape` in

`taypsi/src/Oil/Optimization.hs`. The memoization optimization is `memo` in `taypsi/src/Oil/Optimization.hs`. The driver used in our evaluation is `taype-drivers/emp/taype_driver_emp.ml`. See also [The compilation pipeline].

As the Taypsi syntax presented in the paper uses several typographic conventions (e.g., hat, math symbols, etc.), which cannot be easily reproduced in source code, the definitions in the artifact adopt a slightly different concrete syntax. The following table summarizes the syntactic and naming discrepancies between the Taypsi source code and the listings in the paper.

In paper	In artifact	Comment
`𝟙`	`unit`	Unit type
`𝔹`	`bool`	Boolean type
`ℤ`	`int`	Integer type
`ℕ`	`uint`	Unsigned integer (natural number) type
`×`	`*` or `~*`	Product type former and oblivious product type former
`unsafe fn`	`fn'`	Keyword for defining unsafe functions, i.e. retractions
Name with hat	Prefixed by `~`	e.g., `~list` for `list` with hat
Primitive sections and retractions	`~bool#s`, `~bool#r`, `~int#s` and `~int#r`	
`Ψ`	`#`	Ψ-type, e.g., `#~list` for `Ψlist` with hat
`⟨_,_⟩`	`#(_,_)`	Ψ-type pair
`λ`	`\`	Lambda abstraction, e.g., `\x => ...` for `λx => ...`
`match _ with _`	`match _ with _ end`	Pattern matching
`mux`	`~if`	Oblivious conditional; `mux` in the paper is more consistent with the literatures, but `~if` is more consistent with other oblivious operations in Taypsi

Coq formalization of the core calculus

We have formalized the Taypsi core calculus described in Section 3 in Coq (`~/taypsi-theories`), including proofs of the soundness and obliviousness theorems.

To validate the formalization, run:

```
cd taypsi-theories
make clean
make
```

These commands should output two lines stating `Closed under the global context`. These are generated from the file `taypsi-theories/theories/lang_taypsi/metatheories.v`, indicating that both of the key theorems have been proved without any axioms.

The Coq formalization is also available online; this online version includes a nicely rendered documentation.

The following table summarizes the correspondence between the paper and the Coq formalization:

In paper	In artifact	Notations
Fig. 5	`expr`, `gdef`, `otval`, `oval` and `val` in `taypsi-theories/theories/lang_taypsi/syntax.v`	Defined in the `expr_notations` module in the same file
Fig. 6	`step` and `ectx` in `taypsi-theories/theories/lang_taypsi/semantics.v`	`e -->! e'` (or `Σ ⊨ e -->! e'`) for `step`
Fig. 7	`typing` and `kinding` in `taypsi-theories/theories/lang_taypsi/typing.v`	`Γ ⊢ e : τ` (or `Σ; Γ ⊢ e : τ`) for `typing` and `Γ ⊢ τ :: κ` (or `Σ; Γ ⊢ τ :: κ`) for `kinding`
Fig. 8	`gdef_typing` in `taypsi-theories/theories/lang_taypsi/typing.v`	`Σ ⊢₁ D`
Theorem 3.1 (Obliviousness)	`obliviousness` in `taypsi-theories/theories/lang_taypsi/metatheories.v`	

The `soundness` theorem is also available in `taypsi-theories/theories/lang_taypsi/metatheories.v`.

For simplicity, our mechanization of the core calculus differs slightly from the one presented in the paper:

- The mechanization includes `fold` and `unfold` operations for recursive ADTs (`EFold` and `EUnfold` at `syntax.v`), similar to Ye and Delaware (POPL22), instead of the ML-style ADTs in the paper. The equivalence between these two styles is well-known (cf. Chapter 20 of "Types and Programming Languages").
- Similar to the implementation, the mechanization includes oblivious product and normal product (`EProd` at `syntax.v`, with an `olabel` argument to distinguish them), while the paper has only one product type former for presentation purposes.
- The mechanization uses distinct projections for product and Ψ-type (`EProj` and `EPsiProj` at `syntax.v`), while the paper abuses the notation for presentation.
- The mechanization uses *locally nameless representation* for binders.
- There are some notational differences which should be easy to disambiguate: we use `case .. of ..` instead of `match .. with ..`, for example.

Reproducing the experimental results

To reproduce Figs. 18, 19 and 20 in the paper, you can simply invoke the script that runs all our benchmarks.

```
# At home directory '~'
./bench.sh
```

This script runs each test case 5 times, takes the average of the results, and writes them to the directories `taypsi/examples/output-*`. Finally, this script will execute `taypsi/examples/figs.py` to generate LaTeX tables to `taypsi/examples/figs` for the figures in Section 6 and appendix.

Be warned that this script takes a long time to run: potentially up to 2 hours depending on your machine. You can choose to execute each test for fewer rounds via a command line argument. This would of course produce less accurate results, and it can still take up to 1 hour to run.

```
# Run each test case once
./bench.sh 1
```

You can inspect this script and the scripts it invokes (`bench.sh` in `taypsi`, `taype-pldi` and `taype-sa`) to understand what benchmark suites are tested with what options.

The following table summarizes the correspondence between the generated LaTeX tables and the figures in Section 6. There are also other LaTeX tables generated for the appendix.

In paper	In artifact
First half (list) of Fig. 18	`taypsi/examples/figs/list-bench-full.tex`
Second half (tree) of Fig. 18	`taypsi/examples/figs/tree-bench-full.tex`
First half (list) of Fig. 19	`taypsi/examples/figs/list-opt-full.tex`
Second half (tree) of Fig. 19	`taypsi/examples/figs/tree-opt-full.tex`
Fig. 20	`taypsi/examples/figs/compile-stats-full.tex`

The performance of our benchmarks can vary due to a number of factors, e.g., the specs of the underlying hardware, the cryptographic instructions supported by the CPU, and the overhead of running them in a docker container, so you will most likely not see the exact numbers reported in the paper. Nevertheless, you should observe similar comparative results: Taypsi performs significantly better than Taype on many benchmarks, while doing roughly as well on the remainder.

If you are interested in how the tests are implemented, see The implementation of test cases.

The following tables provide links to the source code of benchmark suites.

List microbenchmark	In `taypsi/examples/list/list.tp`
`elem_1000`	`~elem`
`hamming_1000`	`~hamming_distance`
`euclidean_1000`	`~min_euclidean_distance`
`dot_prod_1000`	`~dot_prod`
`nth_1000`	`~nth`
`map_1000`	`~test_map`
`filter_200`	`~test_filter`
`insert_200`	`~insert`
`insert_list_100`	`~insert_list`
`append_100`	`~append`

`take_200`	`~take`
`flat_map_200`	`~test_concat_map`
`span_200`	`~test_span`
`partition_200`	`~test_partition`

Tree microbenchmark	In `taypsi/examples/tree/tree.tp`
`elem_16`	`~elem`
`prob_16`	`~prob`
`map_16`	`~test_map`
`filter_16`	`~test_filter`
`swap_16`	`~swap`
`path_16`	`~path`
`insert_16`	`~insert`
`bind_8`	`~bind`
`collect_8`	`~test_collect`

Suite in Fig. 20	In artifact
List	`taypsi/examples/list`
Tree	`taypsi/examples/tree`
List (stress)	`taypsi/examples/stress-solver`
Dating	`taypsi/examples/dating`
Medical Records	`taypsi/examples/record`
Secure Calculator	`taypsi/examples/calculator`
Decision Tree	`taypsi/examples/dtree`
K-means	`taypsi/examples/kmeans`
Miscellaneous	`taypsi/examples/misc`

The compilation pipeline

This section discusses how to inspect the different stages of the compilation pipeline.

We use the tutorial `taypsi/examples/tutorial.tp` as a running example, which includes a lot of comments on how to write Taypsi programs. We compile this file by invoking the Taypsi compiler:

```
cd taypsi
# The compiler name is still called taype
cabal run taype -- examples/tutorial/tutorial.tp
```

This command will generate a few files in the `examples/tutorial` directory:

- `tutorial.stage0.tpc` : Taypsi programs in administrative normal form (ANF), with type annotations fully elaborated. However, the typed macros have not been expanded, and the lifting procedure has not been invoked yet.
- `tutorial.lifted.tpc` : lifted programs generated by the lifting algorithm. These programs still contain typed macros and type variables, corresponding to the "lifted functions with macros & type var." block in Fig. 14.
- `tutorial.constraints.sexp` : constraints (Fig. 16) generated by the lifting algorithm, in S-expression format.
- `tutorial.solver.input.sexp` : input to the constraint solver. The constraints generated in the previous step have been lowered to formulas in qualifier-free finite domain theory.
- `tutorial.solver.log` : constraint solver log. It prints out the formulas fed to Z3, statistics information collected by Z3, and each step that the constraint solver algorithm has done.
- `tutorial.solver.output.sexp` : output of the constraint solver. It consists of the type variable assignments for each lifted function.
- `tutorial.stage1.tpc` : lifted programs with type variables instantiated. These programs still contain typed macros, corresponding to the "lifted functions with macros" block in Fig. 14.
- `tutorial.stage2.tpc` : final Taypsi programs. All typed macros are fully elaborated, corresponding to the "well-typed and correct lifted functions" block in Fig. 14.
- `tutorial.oil` : translated OIL programs.
- `tutorial.ml` : translated OCaml programs.

If you want to inspect `tutorial.*.tpc` and `tutorial.oil` to better understand each step in the pipeline, it may be helpful to disable optimizations and print out the programs in a more readable form (as opposed to ANF).

```
cabal run taype -- --fno-opt --readable examples/tutorial/tutorial.tp
```

You can learn about other options by running `cabal run taype -- --help`.

The Taypsi compiler only generates OCaml code as libraries. To make a runnable application, we also have to write the "frontends" which handle I/O and other non-oblivious business. For example, `examples/tutorial/test_elem.ml`, which includes a lot of comments, showcases how we construct a test case as a runnable executable.

We use the Shake build system to streamline the process of building and testing our examples. For instance,

```
# Clean the tutorial example
cabal run shake -- clean/tutorial
# Compile the tutorial example, and its test cases
# --verbose tells shake to print out the commands being run
cabal run shake -- --verbose build/tutorial
# Run all tutorial test cases
cabal run shake -- run/tutorial
# Run an individual test case
```

```
cabal run shake -- run/tutorial/test_elem
# Run a test case with a specific driver (supported drivers are emp and plaintext)
cabal run shake -- run/tutorial/test_elem/plaintext
# See the supported options and targets
cabal run shake -- --help
```

The implementation of test cases

Each of our test cases is implemented as a `test_<name>.ml` file, e.g., `examples/tutorial/test_elem.ml`, which is compiled to an executable. These executables take two arguments (driver and the participating party), and read inputs from `stdin`. Sample input is available for the tutorial example, and we can run these executables through the `dune` build system for OCaml.

```
cd taypsi
# Compile the tutorial example first
cabal run shake -- build/tutorial
cd examples/tutorial
# Run the test case with the plaintext driver.
# This driver only supports one party "trusted".
dune exec ./test_elem.exe plaintext trusted < test_elem.input
# Run the test case with the emp driver (based on EMP toolkit).
# It is a two-party computation with alice and bob.
dune exec ./test_elem.exe emp alice < test_elem.alice.input &
dune exec ./test_elem.exe emp bob < test_elem.bob.input
```

The output of these executables is the collected performance statistics. For the plaintext driver, the output is the number of MUXes performed. For the EMP driver, the output is the running time in microseconds.

As we are testing the oblivious `~elem` function, the input specifies the public view, the private list from Alice, the private integer from Bob, and also the expected result. For example, the file `test_elem.alice.input` is:

```
public: 10
alice: (3 4 7)
bob:
expected: false
```

See the comments in `test_elem.ml` for more details. Note that the value of `bob` is absent (which is 6 in `test_elem.bob.input`), since this is the input to the party Alice.

The actual inputs for the test cases are organized in a CSV file, e.g., `examples/tutorial/test_elem.input.csv`. The first line is the header, specifying which party the data comes from, and then each line specifies a test input. For example, the header of `test_elem.input.csv` is `public,alice,bob,expected`, while one of the test line is `10,(3 4 7),6,false`. The test runner will launch the test programs for each party and feed them the corresponding input. The output is then collected into another CSV file, e.g., `examples/output/tutorial/test_elem.emp.output.csv`. We can invoke the test runner by:

```
cabal run shake -- run/tutorial
```

Installing dependencies and building the source code from scratch

If you want to install the dependencies and build this project on your own machine, you can check out the `README.md` files under `taypsi` and `taype-drivers` directories. Alternatively, the docker file used to build this docker image is also available (`~/Dockerfile` in the docker container or on Zenodo).

Reusability Guide

The Taypsi type checker and compiler (`~/taypsi`) and the Coq formalization of the Taypsi core calculus (`~/taypsi-theories`) should be evaluated for reusability.

The tutorial example (`~/taypsi/examples/tutorial`) contains extensive comments on how to write Taypsi programs and oblivious types (`~/taypsi/examples/tutorial/tutorial.tp`), and on how to use the generated OCaml libraries (`~/taypsi/examples/tutorial/test_elem.ml`). You can play with this example by adding new functions, lifting functions against different policies, and implementing test cases for the generated private functions. You can also follow the larger examples (e.g., `dating` and `record`) and implement a new case study.

The Coq formalization has inline documentation (`coqdoc`); you can also generate rendered documentation by running:

```
cd taypsi-theories
make html
```

A pre-rendered, online version of this documentation is also available.