# Obliv-C

This language was designed to provide a convenient bridge between users of secure computation protocols and its developers. The idea is that “users”, or programmers who want to use this technology but are not experts in cryptography, should be able to use this easily. On the other hand, experts should be able to provide extra primitives as simple library functions without having to invent a full language/compiler from scratch. This is intended to rectify the current situation where a new language needs to be invented every time someone decides to show off (or benchmark) a new cryptographic protocol (e.g. ORAM for random access, private set intersection protocols, or protocols for other adversarial models). This way researchers can test new cryptographic techniques without having to be experts in language design/compilers.

The way this language is supposed to be used is as follows: we start with a code written in this new language we call Obliv-C. All parties agree to execute this program in a secure execution protocol, where most inputs are private, and only the output of the computation are to be revealed to all (or some) parties. The intermediate results stay hidden.

## The Keyword `obliv`

Variables that are supposed to depend on unknown inputs in any way should be declared with the `obliv` qualifier. For example:

```
obliv bool f(obliv int x, int y) {
    return x>y;
}
```

Here, `x` is some variable that depends on unknown inputs, while `y` is a value known to all parties. The result, therefore, needs to be qualified with `obliv`. Non-`obliv` values can be implicitly converted into `obliv`, but never the other way around, unless all parties agree to execute a revealObliv function:

```
int x=4,y;
obliv int z;
z=x; // ok
y=z; // not ok, cannot convert from ‘obliv int’ to ‘int’
revealOblivInt(&y,x,0); // ok, uses revealObliv
```

At this point, `y` is a normal integer with the same value as x, which can now be printed out, written to a file, etc. Here 0 refers to broadcast: all parties executing will receive the value of `x` in their respective copy of `y`. If the second parameter was, e.g. 1 instead of 0, only party 1 would receive the value of `x` in its copy of y --- all other parties would receive some default value (perhaps 0).

## ‘`obliv if`’ Control Structure

The underlying idea is that only the designated output (indicated by revealObliv) should be revealed, and all the intermediate values should be hidden from everyone. While the simple type system above can prevent data dependencies, preventing control dependency is much

harder. In general, we cannot hide the length of time a protocol takes to run, and in the same vein, the number of iterations of a loop is also known to all parties (since they are all executing the protocol). As a result, the language will not allow an obliv variable to be ever used in the condition of any existing control structure (e.g. for, while, if etc. will all need non-obliv variables in their condition). Normally, variables like size of array are publicly known, so you can write things like

```
for (i=0; i<n; i++) …
```

The only exception to this rule is the new control structure 'obliv if', whose syntax is exactly like the normal if structure:

```
obliv if(...) {
  ...
}else obliv if(...) {
  ...
}else {
  ...
}
```

This control structure allows you to use obliv variables in `if` conditions, and works a lot like normal `if` blocks. However, while it allows you to use obliv variables as a condition, you must realize some obvious differences. For one, during execution, nobody would actually know if the condition is true or not --- that is the whole point of obliv-qualified variables. So the blocks will be actually executed regardless of the condition: it doesn't matter if the condition is true or not. So consider this:

```
obliv int x,y;
...
obliv if(x>0) y=4;
```

Over here, *something* will be executed regardless of whether x is positive or not. However, none of the parties would know if their execution actually changed the value of y or not. If x was positive, y is now 4 after this is executed. If not, y still has the same value as before. But they won't know since y is an obliv variable. The assignment operation simply gets disabled if the condition is false.

While obliv if allows you to use obliv variables in a condition, it does come with its own set of restrictions. For example, this is not legal:

```
int x=0;
obliv int y;
obliv if(y>0) x=1; // error x is not obliv
```

In general, inside a given obliv-if (or else) block, you cannot modify any non-obliv variable that was declared outside the block, since that would betray the value of obliv variable `y`. If you find yourself wanting to do this, you probably wanted `x` to be an obliv int in the first place. However, this is legal:

```
obliv if(...) {
  for(int i=0;i<n;i++) { … }
}
```

Even though both i and n are non-obliv, this is okay, since i was declared inside the obliv-if block

and n is never modified. This way, any changes made to non-obliv variables while inside an obliv if scope are never visible from the outside.

## Restrictions on break/continue/return/goto

TODO break/continue/return/goto/case labels etc. cannot be used to jump across 'obliv if' boundaries: jump-like statements can only move control flow within the same obliv-if block.

## Functions

Consider the following code:

```
int x;
void func() { x++; }
…
void protocolMain() {
  obliv if(...)
    func(); // Not valid code!
}
```

This is obviously problematic, considering everything we have discussed about what can or cannot be done inside obliv-if blocks. Here, we are using a function to bypass the restrictions on obliv-if: we are modifying a global, non-obliv variable x inside obliv-if. The language resolves this by introducing obliv functions. We now have two kinds of functions:

```
void func() { … }
void func2() obliv { … }
```

The difference is simple: you are allowed to perform any operation inside a non-obliv function (e.g. `func()`), but it may not be invoked from inside an obliv-if block (or from other obliv functions). On the other hand, `func2` is an obliv function --- it can be called from anywhere, but may not modify any non-obliv variables not declared inside the function.

There is one more issue related to calling functions that comes up when passing parameters by reference. Consider this:

```
void func(int* p1, const int* p2) obliv { … }
```

When this function is being invoked however, there are restrictions on what p1 can point to. For example, this is problematic:

```
int n,m;
obliv if(...) { func(&n,&m); } // error
obliv if(...) { int i; func(&i,&m); } // ok
```

When compiling, the compiler sees n being passed by reference. So it assumes that func may be used to modify n. The variable i is declared inside, so the second invocation is okay, since i can be modified in this scope. In general, non-obliv variables declared outside becomes something like a const variable when entering an obliv scope. We say "something like a const" because there is a slight difference between C's notion of const, and our notion of const here.

## The Keyword `frozen`

As just explained, non-obliv variables declared outside obliv-if does not quite become `'const'`

in an obliv scope. In fact, it becomes `'frozen'`. However, for simple variables, frozen may be implicitly converted to const, e.g. `'frozen int*'` can be used whenever a function really expects `'const int*'`. The difference, however, shows up in case of structs, where they are not implicitly convertible. Here, `'frozen struct foo*'` refers to the "deep-const" version of `foo`. Consider the following code fragment:

```
struct S { int x, *p; };
void func(const struct S* a, frozen struct S* b) {
  // Neither a->x nor b->x can be modified here, as expected
  a->x = 5;  // error
  b->x = 5;  // error
  // But *b->p is different from *a->p
  *a->p = 5; // ok: a->p has type int *const,
             //   as opposed to const int*
  *b->p = 5; // still error; frozen is recursively applied
             //   to all pointer fields in a struct/union
}
```

This slight difference is the reason for us introducing a new keyword `frozen`. In general, any non-obliv variable becomes `frozen`-qualified when entering an obliv scope (obliv-if or obliv functions). For simpler types, this distinction between `const` and `frozen` does not matter. But for structs, unions, pointer-to-pointers etc. it does (e.g. `int **frozen` is actually the same as `const int *const *const`, which is different from `const int**` or `int **const`).

## Advanced Feature: Unconditional Segments

This is the one part of Obliv-C that looks very different from anything in C. Essentially, it chalks out a block of code for unconditional execution even though the surrounding code is conditional on some obliv data:

```
int n = …;
obliv if(cond) { // line 1
  n = 5; // error: cannot modify outer non-obliv n in obliv scope
  ~obliv(c) {
    n = 5; // valid: but the assignment happens even if c is false
  }
}
```

Since the execution of this block does not depend on any obliv value, it gets rid of all restrictions on `frozen` variable modification. The restrictions can be later reinstated by using a nested obliv-if structure, if desired. However, be careful here. Remember that we are in an obliv-if block, and something is probably being executed even if the condition on line 1 is `false` (only that executions will usually have no semantic effect on obliv variables). So you might see n being modified inside `~obliv` even if the condition is `false`. However, this condition can still be detected by checking the new variable named `cond`. In general, the `~obliv(varname)` syntax also declares a new variable `varname` of obliv bool type that will indicate whether the enclosing obliv-if condition is `true` or `false`. This syntax is useful for library developers when they know that there is a more efficient way of executing obliv functions than what one can

otherwise write. Many of the standard library functions we plan to develop will actually use this feature, so you can find useful examples of its use there. In general, when this is used, it is the responsibility of the programmer to make it look like nothing is being executed even when the condition is `false` (ie. its execution should not have any visible non-obliv side effects if cond is `false`, or such side-effects should be constrained to implementation-internal variables only). On the other hand, neither should the programmer assume that it will always be executed even if the condition is `false`, since the compiler is allowed to optimize it away.

## Implementing a Protocol

need a section that explains:

```
feedOblivInputs,
setupObliv,
revealObliv

ocBroadcastInt
```