OCTAL: Data Oblivious Programming Language

Biniyam Mengist Tiruye
University of Michigan
btiruye@umich.edu

Lauren Biernacki
University of Michigan
lbiernac@umich.edu

Todd Austin
University of Michigan
austin@umich.edu

***Abstract*—During the 1970s, a curious set of algorithms called data oblivious algorithms started to catch the attention of security research, because of the numerous applications they provided due to their unique properties. In particular, data-oblivious algorithms execute independently from their input data. This means that an attacker can't learn anything by observing how these algorithms execute, as these algorithms produce no memory or control side channels, thereby protecting private data against various attacks. Programmers, However, avoid these algorithms because they require a highly stylized form of programming, resulting in a very error-prone design and implementation process, in addition to the fact that data-oblivious programs are much less efficient than their native counterparts. To address these problems, we present OCTAL (Oblivious, Constant Time, and Adaptable Language), a high-level domain-specific language that automates the design and implementation of data-oblivious programs. OCTAL makes the development of data-oblivious algorithms effortless by moving data-oblivious transformations into the compiler. OCTAL also facilitates the development of more efficient data-oblivious algorithms by providing the environment to easily explore algorithm design spaces with its familiar C++-like interface. We evaluate Octal using workloads from the VIP-Bench Benchmark suite and show that OCTAL achieves an average 3× improvement in development time and an average 37% reduction in lines of code over mechanical data-oblivious conversion.**

I. OCTAL

A. Overview

OCTAL is built on top of C++ with a minimal difference from traditional C++ syntax and semantics. Private values are declared using OCTAL's private data types and data oblivious transformations are performed on these private types via a source-to-source transformation of the OCTAL program using Clang. The only change a programmer needs to make in order to write an OCTAL program is to specify private data using these private data types. OCTAL provides custom private types for all primitive data types.

B. Clang

Clang is a front-end for the C and C++ programming languages that includes a pre-processor and Lexer, which generates tokens. These tokens are then processed by a parser and semantic analyzer, resulting in an Abstract Syntax Tree (AST) which is an equivalent representation of the source code. The AST is mostly immutable and it preserves everything from the source code except comments and some formatting information. Therefore, it is a managed representation of the source code and enables easier source-to-source transformations from the AST. Finally, the code generation path converts the AST into LLVM IR. The Clang AST consists of three main base classes: type, statement, and declaration classes. The type class represents data types in the language, including built-in types, pointer types, and array types. The statement class represents different statements in the language, such as if statements, for statements, and return statements. The declaration class represents different kinds of declarations present in the language, including variable declaration and function declaration. The AST is composed of one of these classes or their derivatives. Given this structure, our implementation idea is to traverse the AST and provide a transformation for each class encountered in the tree.

C. Source-to-source Transformation

In the compiler, the OCTAL code is transformed into an AST representation of the program and traversed through to identify statements to be transformed into data-oblivious implementations. Finally, the transformed program is generated as a C++ program that is independent of the underlying representation of any Privacy Enhanced Computation technology.

1) Declaration Statement Transformation: In order to preserve the secrecy of private values, OCTAL enforces obliviousness rules on variable declarations and assignments. Specifically, OCTAL traverses through each variable declaration and assignment and checks whether they violate these rules by, for example, assigning a private variable to a non-private one. If that is the case, OCTAL will transform the non-private variable into the corresponding private type.

2) Conditional Statement Transformation: If-conversion serves to address conditionally-executed instructions to ensure that a workload's execution is independent of its input data. Specifically, if-statements with input-dependent conditions violate obliviousness as they introduce data-dependent control flow in the resultant binary. To make these statements data-oblivious, they must be transformed through if-conversion, which removes the control dependencies by predicating the instructions within the statement's basic block. OCTAL achieves this by using the x86-64 CMOVcc primitive to guard statements inside a condition by their corresponding predicates.

3) Loop Transformation: Loop statements with input-dependent iterations similarly violate obliviousness as they introduce data-dependent branches. Specifically, loops with a variable number of iterations or early-exit conditions must be transformed.

OCTAL uses a heuristic that, when encountering a loop with input-dependent iterations, converts the loop to take a fixed number of iterations. Any condition that previously terminated the loop is transformed into a Boolean value that predicates writes to any variables live outside of the loop. If the loop's termination condition is not dependent on a secret variable then no transformation is needed.

4) Access Transformation: Input-dependent memory access patterns violate obliviousness. These memory patterns often manifest as array accesses where the index depends on input data. OCTAL checks whether memory accesses violate obliviousness rules and designate private accesses with an interface that can be implemented by the underlying PEC. One such implementation is replacing data-dependent array accesses with reads and writes to the entire data structure while only performing the desired update for the index specified by the non-oblivious operation.

5) Return Transformation: Programs with return statements based on a secret variable also violate obliviousness as they introduce time variations based on the input value. In order to mitigate this, OCTAL defines a private return predicate at the top of the function and initializes it to false, this predicate guards the execution of all statements in the function. A return_value variable of the function's return type is also defined and set to a zero value. Then whenever a return statement is detected the value is copied to return_value and the return predicate is updated.

6) Break and Continue Transformation: Break and continue statements can also violate obliviousness by causing loops to have variable iterations. To handle break statements, OCTAL defines a break predicate for each loop and uses it to guard all statements in the loop body. This predicate is initialized as false and it is updated to true whenever break statements are detected. Continues are handled the same way except continue predicates are defined inside the loop.

II. EVALUATION

A. *Workloads*

We evaluate OCTAL using workloads from the VIP-Bench benchmark suite . The VIP-Bench suite is instrumental for assessing the effectiveness and efficiency of Privacy Enhanced Technologies (PETs), as it provides both native and data-oblivious variants of workloads for testing. For our evaluation, we select four of the native VIP-Bench workloads and convert them into OCTAL implementations. These workloads require different types of transformations and consist of different types of operations.

B. *Results*

Our investigation into the performance of OCTAL in comparison to the mechanical conversion of VIP-Bench workloads to data-oblivious variants has demonstrated a considerable advantage in terms of both lines of code and development time. Specifically, we found that OCTAL achieved an average reduction in lines of code of 37%, while also providing an average 13,000x speedup in conversion time. This speedup in conversion time facilitates the development process up to 3x on average. These findings indicate that OCTAL is an effective tool for simplifying the development of data-oblivious algorithms. Figures 1 and 2 visualize the Lines of Code reduction and Development Time speedup for each workload.

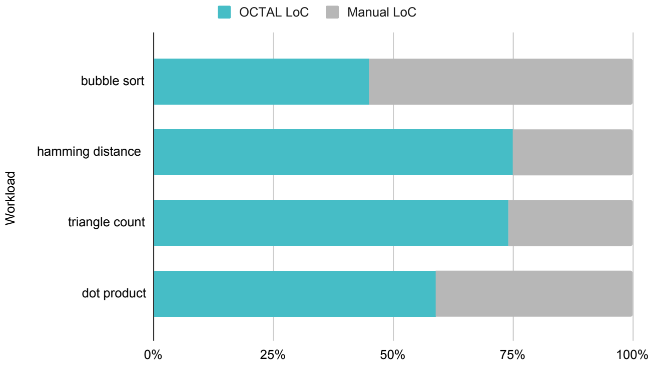

Fig. 1: **Lines of code, Relative to Manual conversion**

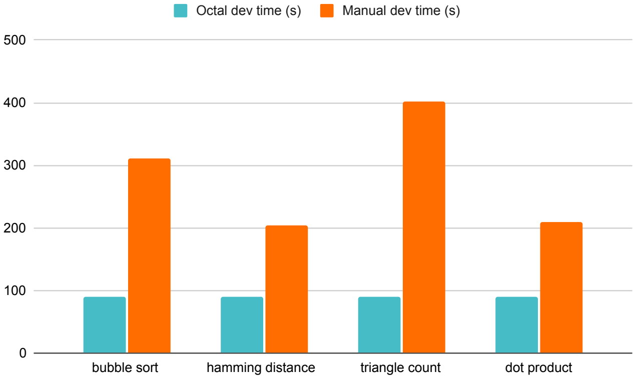

Fig. 2: **Development time, Relative to Manual conversion**