Smaug: Modular Augmentation of LLVM for MPC

Radhika Garg
Northwestern University
radhikaradhika2028@u.northwestern.edu

Xiao Wang
Northwestern University
wangxiao@northwestern.edu

***Abstract*—Secure multi-party computation (MPC) is a crucial tool for privacy-preserving computation, but it is getting increasingly complicated due to recent advancements and optimizations. Programming tools for MPC allow programmers to develop MPC applications without mastering all cryptography. However, most existing MPC programming tools fail to attract real users due to the lack of documentation, maintenance, and the ability to compose with legacy codebases. In this work, we build Smaug, a modular extension of LLVM. Smaug seamlessly brings all LLVM support to MPC programmers, including error messaging, documentation, code optimization, and frontend support to compile from various languages to LLVM intermediate representation (IR). Smaug can efficiently convert non-oblivious LLVM IR to their oblivious counterparts while applying popular optimizations as LLVM code transformations. With benchmarks written in C++ and Rust and backends for Yao and GMW protocols, we observe that Smaug performs as well as (and sometimes much better than) prior tools using domain-specific languages with similar backends. Finally, we use Smaug to compile open-source projects that implement Minesweeper and Blackjack, producing usable two-party games with ease.**

1. Introduction

Secure multi-party computation (MPC) allows multiple parties to jointly evaluate a function while keeping their individual inputs private. Due to the complexity of MPC protocols, one important task is to build tools that can facilitate the development of MPC-based applications. This is often a complicated task. First, programs executed in MPC must be trace and data oblivious, meaning that the program counter and data access shall not reveal parties' private information. Second, the cost model of MPC differs from standard hardware, e.g., x86; thus, different optimizations are needed to enjoy the state-of-the-art MPC optimizations. There have been various approaches to reduce programmers' burden when handling these difficulties.

1) **Bring Your Own Machine (BYOM).** The most generic approach is to work at the assembly level. These works implement backends that can securely emulate the execution of assembly code directly [WGMK16], [SZD+16], [Kel19], [YPHK23]. In this approach, the programmer can use any existing language and compiler to obtain the assembly code of the program, and then execute it on MPC-emulated backends. The overhead for programmers is minimal — they do not even need to know that the code will be executed over MPC. However, the execution overhead is often high because the backend must hide the instruction type and data access in each cycle. In particular, this approach requires using oblivious RAM (ORAM) to emulate every instruction, leading to high overhead.
2) **Bring Your Own Compiler (BYOC).** There are also works that implement their own compilers but use syntax from existing languages [HFKV12], [BDK+18], [HST+21], [LIS+23], [CZO+23]. Such an approach allows the programmer to keep using their favorite languages; the custom-built compiler would transform the source program to something the MPC backends can use, bypassing legacy compiler toolchains and machine abstractions altogether. This approach is often more efficient than BYOM, as the compiler can use program structures to optimize the MPC execution. However, it has some other drawbacks. First, many compilers can only handle toy examples rather than large projects and do not emit any error messages/hints about compilation errors. This is because most academic-built compilers do not have the resources to support a compiler toolchain. In addition, it is challenging to integrate MPC code with cleartext code, even if they are in the same language. The MPC code is compiled to cryptographic backends, while cleartext code is compiled normally.
3) **Bring Your Own Language (BYOL).** Many prior works focus on designing a domain-specific language [ARG+21], [MGC+16], [YD23], [CGR+19], [Kel20] or additional language syntax [ZSB13], [ZE15], [LWN+15], and an associated toolchain such that the programmers can learn and use. This approach has the best flexibility in designing annotations and syntax to capture the program's structure. However, the programmers have to learn new language features, and it also has drawbacks similar to BYOC since existing compilers and machine abstractions are bypassed in this case as well.

There are also other frameworks [WMK16], [HEKM11], [SHS+15], [BDST22] that build "libraries" for programmers to use, but they push even more burden on programmers to figure out how to convert the program to something in the library calls while ensuring obliviousness and high efficiency.

What About LLVM? As the most popular compiler infrastructure, LLVM appears to be a great candidate for building a compiler toolchain for MPC. Indeed, it has been discussed by multiple prior works but only on why LLVM is **not** suitable. It turns out that directly reusing the LLVM framework is challenging. For example, COMBINE [LIS+23] argued that important information is missing for ϕ instruction, something in the LLVM control flow graph (CFG) to handle conditional statements. They also argued that the complexity of LLVM makes it challenging to perform control-dependent analysis. Heldmann et al. [HST+21] adapted LLVM to output circuits for MPC applications but requires that all conditional statements be independent of any private information, bypassing the ϕ instruction issue mentioned above. Keller [Kel19] uses LLVM to parse the source language but adopts the BYOM method to execute the LLVM-IR. In this paper, we defy this common belief by presenting Smaug, an augmentation of LLVM for integrated programming support of MPC applications.

1.1. Features of Smaug

Smaug is implemented completely as a set of modules of LLVM — when MPC flags are disabled, the tool compiles normal programs as usual. This is achieved by collecting MPC-related information and performing MPC-related code transformation as "LLVM passes" on the code. Smaug achieves a trifecta of key features by extending LLVM in a modular way rather than building an independent compiler.

1) **Inheriting support for LLVM toolchain.** The core of LLVM focuses on optimizing LLVM-IR itself, but many industry-grade compiler frontends can compile to LLVM-IR, most notably `clang` and `rustc`. By working on top of LLVM-IR directly, Smaug inherits from existing LLVM frontends to support almost any language. At the same time, Smaug can produce meaningful compile-error messages/hints just like other industry-grade compilers.
2) **Modular integration of MPC and legacy libraries.** By producing valid LLVM-IR, where crypto operations are LLVM function calls, Smaug allows linking these functions to MPC implementations using an existing linker (e.g., GNU `ld`). Similarly, any custom optimizations, e.g., table lookups, can be integrated seamlessly. This also means that the dynamic interpretation of the MPC program is done at the native-code level. An additional benefit of extending LLVM is that we allow programmers to use legacy libraries (e.g., `sqlite`) with MPC components seamlessly in one executable. This was only possible with a library-based approach.
3) **Repurposing LLVM optimization for MPC.** Since Smaug is an LLVM module, we have access to existing LLVM optimization modules, many of which are beneficial to MPC. This includes dead-code elimination, CFG simplification, optimizations for vectorization, which minimizes circuit depth, and analysis passes like memory dependence analysis and loop analysis, which help minimize the effort of writing custom optimization passes. Although some prior works manually implemented a subset of them, no framework implements all since they don't build on top of each other. Basing Smaug on LLVM means we can support all of them simultaneously and enjoy all careful optimizations from the LLVM code base. We plan to open source Smaug so that other researchers can add more optimizations; LLVM helps with this goal as it provides a modular infrastructure for extension.

1.2. Technical Contributions

Fully integrating with LLVM is challenging, and we make the following contributions to achieve the best MPC efficiency without burdening the programmers.

Toolchain overview. The core principle of our approach is to use the LLVM-IR as the only intermediate representation throughout the compilation and execution. This way, we can fully enjoy all the benefits of the LLVM toolchain. A program in an existing language will first be converted to LLVM-IR using any compiler frontend. From this, we run some code transformations provided by LLVM and ones provided by Smaug to make the program oblivious and optimized towards MPC (e.g., depth reduction). Finally, we have a valid LLVM IR that contains both cleartext and private computation. From here, we use existing compiler linkers to link cleartext function/library calls to their object files and link MPC operations to existing libraries.

Supporting mixed branching. The LLVM optimization passes can extensively alter the CFG, resulting in CFGs with `goto` statements under private branches, which cannot be easily handled by prior approaches. We design an efficient transformation that can handle such cases even within nested branches, where some have public conditions and some have private conditions. Since all loop syntax (i.e., `for`, `while`, `do-while` loops) are the same within LLVM-IR, as a by-product, we are able to efficiently support all loops recognized by LLVM as long as the number of iterations does not depend on the private values.

MPC optimizations as LLVM passes. To demonstrate the generality of LLVM in the context of MPC, we implement a variety of optimizations that are suitable for MPC. These optimizations either do not require any extra annotation from programmers or only require adding annotations that already appear in existing compilers. In particular, we implement several optimizations from prior works [BHWK16], [GF95], [LIS+23] for circuit size and depth minimization. We demonstrate that by utilizing the extensive analysis modules provided by LLVM, we can implement these optimizations with reduced effort and achieve comparable or better performance. Further, this shows that any future work on circuit optimization for MPC can seamlessly add custom transformation modules to Smaug and use the existing analysis modules in order to minimize development effort.

Comparison with prior works. We compare the performance of Smaug with COMBINE [LIS+23], CBMC-GC [HFKV12], MP-SPDZ [Kel20], and Obliv-C [ZE15]. During the compilation of the benchmarks, Smaug uses computational resources comparable to Obliv-C (which fully relies on the programmers in optimizing the circuit) and orders of magnitude less time and memory than the other prior works. Additionally, the resulting programs for Smaug either outperform prior works or have comparable performance. For instance, when computing convex hull over a polygon with 256 vertices, Smaug is $111\times$ and $28\times$ faster than COMBINE and MP-SPDZ, respectively, with GMW backends, and performs comparably to Obliv-C with a garbled-circuit backend. Section 7.1 and Section 7.2 show a detailed comparison over a rich set of programs.

Real-World Programs. We show that Smaug can compile programs written in both Rust and C/C++. Section 7.3 shows an example program and briefly compares the resulting programs for both languages. Furthermore, we adapt several open-sourced programs with minimal additions to enable secure computation. For instance, we build a K-Means library using an open-sourced plaintext computation implementation in C **without** any change using Smaug. We use the resulting library to classify data shared among two parties in a MySQL database using the standard C++ MySQL library. We also adapt public Minesweeper and Blackjack implementations to a two-party setting and are able to use Smaug for secure computation without additional effort. The average response time for the blackjack implementation, where the dealer runs on 2-pc, and one of the parties is the player, is 3 ms. We describe the detailed examples and their performance in Section 7.4.

1.3. Outline of the Paper

Section 2 presents the preliminary definitions and essential background on LLVM. In Section 3, we provide an in-depth discussion of the design of Smaug. Section 4 and Section 5 describe our transformation algorithms and their implementation. Section 6 explains how Smaug integrates with an MPC backend. Finally, in Section 7, we present the evaluation, including the comparison with prior works and real-world examples.

2. Preliminaries

2.1. Control Flow Graph

A control flow graph (CFG) [All70] is a directed graph, $G = (B, E)$ where $B = \{b_1, b_2, ..., b_n\}$ is the set of nodes, and E is the set of directed edges, each represented by an ordered pair (b_i, b_j) of nodes. Each node in a CFG represents a **basic block**, i.e., a sequence of instructions with one and only one entry point (first instruction) and exit point (last instruction). The edges in a CFG represent the control flow between basic blocks: an edge from block A to block B indicates that execution can pass from the last instruction in A to the first instruction in B. This can happen due to conditional or unconditional branches or the natural fall-through from one block to another.

(Post-)Dominator Trees. Block A dominates B if every path from the entry point of the CFG to B passes through A. The immediate dominator of block B is the last dominator on the path from entry to B. A dominator tree is a directed acyclic graph with an edge from node A to B if A is the immediate dominator of B in the CFG. Similarly, A post-dominates B if every path from B to the exit node of the CFG passes through A. A post-dominator tree is a directed acyclic graph with an edge from node A to B if A is the immediate post-dominator of B.

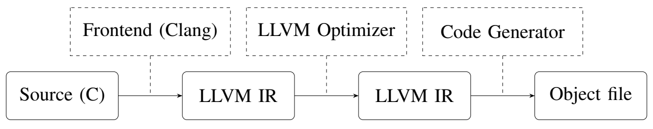


Figure 1: **Overview of the LLVM Compiler Flow.** Frontend (Clang frontend for C) parses the source code and transforms it to LLVM IR. LLVM core libraries are used in the middle end for optimization. LLVM provides libraries for converting the LLVM IR to machine code for several targets.

2.2. LLVM

LLVM [LA04] is a collection of compiler and toolchain technologies that are modular, reusable, and source/target-agnostic. A significant advantage of LLVM is its adaptability, serving as the backbone for various compilers and language front ends, most notably C, C++, and Rust. The backend of LLVM features a target-independent code generator that can create output for several types of target CPUs, including X86, PowerPC, ARM, and SPARC. Figure 1 provides an overview of the LLVM compiler workflow.

The **LLVM intermediate representation (IIVL-IR)** is a strongly-typed assembly-like intermediate representation that serves as a platform-independent way of representing code. It has infinite virtual registers and is a static single assignment (SSA) based representation.

The LLVM optimizer provides various modules for program analysis and transformations to optimize the IR. Its modular structure also allows developers to add their own analysis and transformation modules.

Static Single Assignment (SSA) form. SSA [CFR+91] is a way of structuring a program where each variable is assigned exactly once. This makes the data flow in the program more explicit and simplifies certain optimizations, as it is easier for the compiler to reason about the lifetimes and usages of variables. Figure 3 shows an example program in its non-SSA and SSA-based representation, where x is assigned twice in Figure 3a in $b2$ and $b3$ but once in Figure 3b in $b4$.

Phi (Φ) instruction. When transforming code into the SSA form, compilers face the challenge of which variable to use when a variable can be assigned in multiple paths of a program, especially within loops or conditional blocks. To resolve this, SSA introduces ϕ instructions, a conceptual tool used in the IR to merge different incoming values into a single SSA variable. The ϕ instructions in LLVM IR contain pairs of variables and incoming basic block labels. The variable is chosen by matching the last executed basic block label with the incoming basic block label in the instruction operand. For the program in Figure 3b, $x = x1$ if $b2$ is executed before $b4$ and $x = x2$ if $b3$ is executed before $b4$.

LLVM loop-vectorizer. The LLVM optimizer includes modules that enable vectorization for optimal perfor-

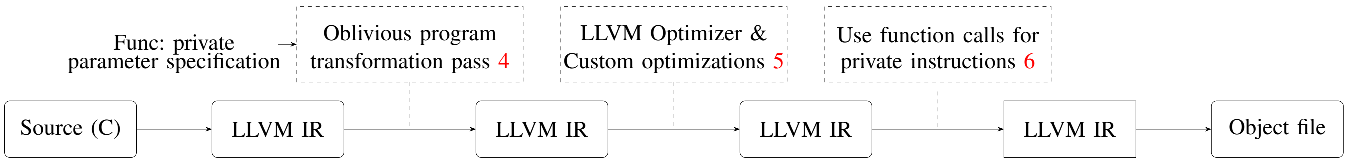
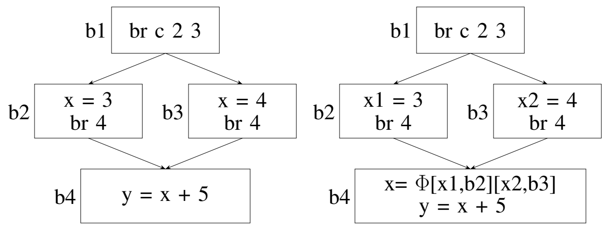

Func: private parameter specification
Oblivious program transformation pass 4
LLVM Optimizer & Custom optimizations 5
Use function calls for private instructions 6
Source (C)
LLVM IR
LLVM IR
LLVM IR
LLVM IR
Object file

Figure 2: **Overview of Smaug.** The front end (Clang) converts the source (C) to LLVM IR. The IR can be optimized, for instance, by using -O1 with Clang. We transform the program (LLVM IR) given the specification for private function parameters for each function in a module into an oblivious program. Further, we optimize the program using the LLVM optimizer and the custom transformation passes we add that are specific for MPC circuit optimization. Finally, we replace the instructions where the input is private with the corresponding MPC function calls. Now, we use the LLVM code generator to compile the program to object code for the respective target.



(a) Non-SSA representation. (b) SSA-based representation.

Figure 3: **Example of SSA form.** Figure 3a shows a program that branches based on a conditional block and assigns x, then depending on the path taken, y is computed. Figure 3b shows that in SSA-based representation, both branches use different variables, and the joining block (b4) has a Φ instruction that chooses from the variables depending on the last executed block.

mance in plaintext computation. The LLVM loop vectorizer [TSS+17] first checks if it is legal to vectorize a loop. Then, it makes vectorization plans and analyses the cost after vectorization. It further updates those plans using several factors like uniform branches, interleaving, etc. Finally, it chooses the best vectorization plan using the cost model. The vectorization plan is executed if it is beneficial against the scalar cost. The vectorization plan uses scalable or fixed-width instructions based on the target support.

2.3. Oblivious Algorithm

Oblivious algorithms are ones whose program execution trace, i.e., the movement of the program counter, is independent of the private inputs. Oblivious algorithms protect side-channel information related to program execution and have found applications in cryptography [GKK+12], constant-time programming [RLT15], and cloud computing [SZE+16]. Once a program is represented in a trace-oblivious manner, producing a circuit that computes the same function is straightforward. The circuit size can be captured by the complexity of the oblivious algorithm. Thus, many works in optimizing circuits for MPC boil down to finding better oblivious algorithms. Any program can be compiled to its oblivious counterpart using oblivious RAM, but one could obtain better efficiency through more optimizations.

3. Smaug Design

Most MPC compilers design their own language/IR, thus suffering from various drawbacks, like lack of maintenance, difficulty in integrating MPC code with cleartext code, etc. To fill this gap, we propose Smaug. Smaug is built on top of the LLVM toolchain and works by adding custom transformation modules to the LLVM optimizer. Therefore, our framework can compile source code written in any language that can be compiled into the LLVM IR. Figure 2 shows a brief overview of our compiler, which we explain below.

3.1. Support Non-Oblivious Programs

As the first step, we use existing off-the-shelf tools (e.g., `clang`) to compile the source code to LLVM IR and then use the LLVM optimizer to perform optimizations that are useful for MPC, including dead-code elimination, combining redundant instructions, etc. Then, we need to transform the program to be oblivious. There are several approaches to handling non-oblivious programs: 1) obliviously emulate every instruction in the program using ORAMs [WGMK16], [SZD+16], [YPHK23]; 2) assume some restricted behavior of the source code and perform more efficient transformation of the code. Such restrictions may assume that the program is branchless [HST+21] or assume that all the branches are based on private conditions and all the loop bounds are known at compile time [LIS+23], [HFKV12]. In practice, these restrictions are captured using customized language features [LWN+15], [ZE15], [ZSB13], [MGC+16], [CGR+19] or by simply reporting an error when the assumption is not met. The goal of Smaug is to work on top of unmodified LLVM-IR and embrace its generic CFG as much as possible with minimal assumptions on the input programs.

Firstly, to identify the private variables, we input a specification that specifies the nature of input parameters and the output for each function. Using this specification and the program compiled to LLVM IR, we analyze the LLVM IR to identify if the control flow depends on some private variable and then transform it into an oblivious program. The basic idea for the transformation is to execute a basic block regardless of the private condition expression associated with it and nullify the effect of this execution over the program if the corresponding condition expression evaluates to false. So far, it is similar to prior works, but one unique challenge in our case is supporting arbitrary CFG. Traditionally, DSLs restrict the source language so that the CFGs are nicely structured. However, we take as input unmodified LLVM-IR, potentially after some optimizations, including CFG simplification, and thus cannot assume much structure. For instance, a code block may be reachable from two paths, one with a public condition and one with a private condition. In this case, the execution condition of this block is a disjunction of public and private variables. One could assume that all conditions are private and transform the program accordingly, but this would produce suboptimal results. In Smaug, we design a new code transformation module to convert generic mix-condition LLVM-IR to its oblivious counterpart efficiently. We discuss details in Section 4.

We note that LLVM does not recognize irreducible loops as loops because they are unsuitable for specific transformations and optimizations due to the non-trivial dominance relationship among the blocks in the irreducible loop. Thus, we omit any discussion for CFGs that contain irreducible loops and assume that the dominance relationship among all the basic blocks in the CFG is trivial. Furthermore, we describe how to handle private loops without any addition to the parser of the high-level language in Appendix A.6.

3.2. Circuit Optimization for MPC

Once we have an oblivious program, we can optimize it further while maintaining the obliviousness. Several works deal with circuit size optimization for MPC, including loop vectorization [LIS+23], depth reduction using parallelization techniques [BHWK16], and loop coalescing [GF95]. Each of these works only implements one or two optimizations, and it is impossible to enjoy all of them without reimplementing everything. The goal of LLVM is to modularize optimizations for ease of extension, and we observe that it can also serve MPC applications in two aspects. First, many LLVM optimizations can be applied directly to MPC programs to reduce the size of the functions. This includes Dead Code Elimination (`dce`), Combine Redundant Instructions (`instcombine`), Sparse Conditional Constant Propagation (`sccp`), etc. Prior works implement some of them, but Smaug is equipped with these optimizations for free. Second, we add our own code transformations to optimize the code further. LLVM has been viewed as overly complex for MPC, but in this paper, we show that the existing LLVM code analysis framework can actually help implement optimizations. We show how three important optimizations proposed in prior works can be easily implemented in LLVM in Smaug. We further show that we achieve comparable performance to special-purpose compilers. Below, we describe these optimizations and our approach in brief.

Vectorization. Some MPC protocols like GMW [GMW87] require one or more rounds of communication for each layer of circuits. Therefore, MPC libraries implementing these protocols usually offer an interface to use SIMD gates to allow the most effective use of the protocol. To fully utilize such protocols, one would need to identify the blocks of code that can be made SIMD and potentially rewrite the program for optimality. In prior works, it is achieved by writing their own optimizations. We observe that LLVM already provides the core optimization modules for this task, which can be used seamlessly in conjunction with other optimizations.

In particular, the LLVM optimizer includes modules to identify as many vectorization opportunities as possible, using Loop Vectorizer and Superword-Level Parallelism Vectorizer to fully use SSE-type instructions and instruction pipelining. The vectorizer takes into account the cost of instructions before and after vectorization. By having the MPC protocols in valid LLVM IR, we are able to use these modules directly. To maximize the level of vectorization, we make the following changes: First, we force the vectorizer to use scalable width vector instructions for loops with MPC instructions. Second, we write a transformation pass that splits a loop containing vectorizable and non-vectorizable operations into multiple loops, potentially with increased memory. This is needed because LLVM, by default, does not vectorize such mixed loops due to memory concerns. Fortunately, LLVM already provides analysis modules, including loop access analysis and scalar evolution analysis, to obtain simple information about the loop, like loop count and induction, to complex information about memory dependence, all of which help identify vectorizable operations in a loop and write the loop splitting transformation pass. By comprehensive LLVM-based vectorizer, we find that Smaug can capture almost all SIMD opportunities. Section 5.1 describes the LLVM vectorizer and the modifications we do in detail.

Parallelization of reduction patterns. Reduction patterns in a program are computational patterns that combine multiple values into a single result, like adding all elements in an array or finding the maximum value in an array. These computational patterns can typically be parallelized by computing the operation in a tree-based fashion, where the operations at each level of the tree can be vectorized. In [BHWK16], the authors implement this optimization as part of CBMC-GC [HFKV12] to produce low-depth circuits.

We find that the vectorization passes of LLVM find such reduction patterns in vectorizable loops and even outside of loops (SLP vectorizer) and replace the computation with **vector.reduce** intrinsic function calls. Thus, in our loop-splitting transformation pass, we use the LLVM vectorizer's analysis to identify the reduction patterns and separate the reduction instructions from any non-vectorizable instructions. Now, to parallelize the reduction operations, we simply need to add a gadget in the MPC library that uses a tree-like structure to compute the result and replace the intrinsic call with a call to the corresponding gadget function.

Loop coalescing. Loop coalescing is a transformation pass that is useful when dealing with nested loops where the inner loop's count is private, but the total number of iterations is known and public. For example, when using an adjacency list as input to Dijkstra's algorithm, we do not know the out-degree of each vertex, but the total number of edges is public. ObliVM [LWN+15] proposes a DSL that allows a programmer to specify the max loop count. This information is used to transform a nested loop into a single loop for optimal performance, such that if the outer loop is executed at most n times and the inner loop is executed at most m times, then the resulting program achieves $O(n + m)$

performance.

Our work adapts this transformation for LLVM-IR, achieving it without the need for additional syntax. To avoid introducing a new pragma in Clang, we use loop unroll_count to obtain the max loop count for nested loops where the innermost loop has a private exit. We describe how we implement our adaptation in ≈ 500 lines of code with the help of LLVM's utility functionalities in Section 5.3

3.3. End-to-End Integration

MPC compilers typically compile the input program into a specific circuit format, like the Bristol format, which the MPC backend libraries can interpret. In [BHK+23], Braun et al. describe that these formats often lack one or more features like vector instructions, loops, and function calls and propose FUSE-IR to bridge the gap. However, the proposed format, FUSE-IR, does not support control flow elements like branch instructions or loops where the loop bound is determined at runtime. Furthermore, their framework lacks the extensive program analysis and optimization modules offered by LLVM. Note that a constant loop bound allows loop unrolling, and the program, in turn, does not have any control flow elements. The lack of support for loops requires the programmer to recompile the program every time the number of elements changes, such as in biometric matching over n data samples.

Smaug works on top of LLVM; thus, it supports control-flow elements and allows the program to be dynamic. This dynamism is not supported by any of the final circuit formats used in MPC. Thus, to hook the program with an MPC backend, we create an MPC interface similar to the LLVM instructions and replace the instructions that operate on private inputs with the corresponding function calls to this interface. We define a mapping of the instruction code and input data type with the corresponding MPC function signatures to determine the function to call. Given the mapping of the instruction code and corresponding MPC function signatures, this allows us to hook the program with any MPC library. Section 6 describes how we use the MPC library for the required secure computation.

4. Transformation To Oblivious Programs

This section presents our algorithm that transforms non-oblivious programs into oblivious ones. The source program typically includes constructs like `goto`, `if else`, and `for` loops, resulting in a control flow dependent on the runtime values of variables. These variables can be public or private, depending on the condition variable of the `if else` constructs or the loop-exit condition. In prior works, this is resolved either by BYOM, which completely hides the program trace with high cost, or BYOC(L), which restricts the behavior of conditional constructs to only well-behaved ones, e.g., no `goto`. Such an assumption may no longer hold, especially after LLVM optimizations. Therefore, we propose a generic approach to compiling a non-oblivious program into an oblivious program, covering essentially all common cases.

As mentioned earlier, our transformation algorithm takes as input a specification for each function to determine the associated private parameters and the nature of the function output. Using this specification and the input program, we perform a simple taint analysis to label the instructions that are dependent on some private information. The terminator instruction in the LLVM IR determines the control flow. We assume the terminator instructions can only be return (`ret`) or branch (`br`) instructions. When analyzing the CFG at the function level, `ret` is the terminator of the exit block, and all other blocks have either conditional or unconditional `br` instructions as the terminator. The conditional `br` instructions can depend on a public (known to all parties at runtime) or a private variable. Formally, the CFG of a function is a graph G where each block has a maximum out-degree of two, and the edge (b_i, b_j) from block b_i to b_j can be private or public depending on the branch condition of b_i. If the branch condition at the terminator of block b is private, then the choice of the next block to execute leaks the value of the condition variable at runtime. Thus, we need to transform the program so that it does not contain any private edges while maintaining correctness.

Here, we assume that all loops in the program have a public loop-bound and defer how to handle loops with private loop-bound to Appendix A.6. We first show an example transformation and then describe each step in detail. The proof of correctness is deferred to Appendix A.5.

The obvious idea for transformation is to assume all branch conditions are private and transform it into a straight-line program. To maintain the dominance relationship of each value from its users, the straight-line program must preserve the topological order of the basic blocks. However, undesirable execution of a basic block can lead to incorrect output. Given that LLVM IR is SSA-based, the undesirable execution of a basic block only affects the program state because of its `store` instructions and the values propagated via ϕ instructions. Thus, we describe in detail how to update the `store` and ϕ instructions in Appendix A.3 and Section 4.2, respectively. Note that topological order is defined for directed acyclic graphs, but a CFG might contain cycles; thus, we discuss how to obtain the reverse post order to handle loops in Appendix A.2. Furthermore, simply transforming into a straight-line program results in suboptimal performance. Section 4.1 presents a brief example program showing how to produce an optimal transformation of the original program.

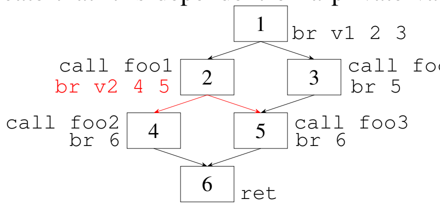
4.1. High-level Idea

Here, we illustrate an example program that contains mixed-branch conditions and discuss its transformation into an oblivious program w.r.t. any private variables.

```
void test(bool v2, bool v1) {
    if (v1) {
        foo1();
        if (v2) foo2();
        else    foo3();
    }
    else {
        foo4();
```

```
            foo3();
        }
}
```

In the above program, v2 is private, while v1 is public. After certain preliminary optimizations of LLVM, the following CFGs could be the result, where red instructions indicate that it is dependent on a private value.

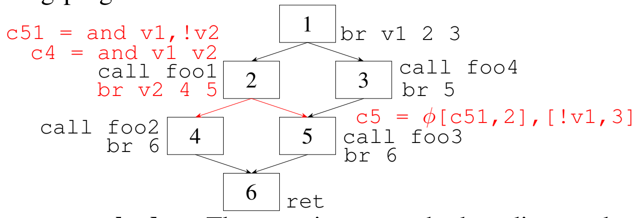


In practice, the CFG could be more complicated for complex programs, but we use this to illustrate our main idea. In the final CFG, we want all the branch instructions to be independent of private variables (v2 here).

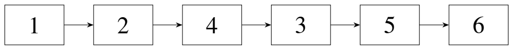
Find the condition expression for each block. To determine the condition expression for each basic block, we note that the entry block always has the condition true. For all other blocks, the condition is chosen from a set of conjunctions of the branch condition and the condition expressions of its predecessor blocks depending on the predecessor block executed last. The table below summarizes the condition expressions for each basic block:

Block	Condition Expression	Block	Condition Expression
1	true	4	v1 & v2
2	v1	5	{(v1 & !v2), !v1}
3	!v1	6	true

To find the most compact expression, as in the above table, one would need to perform more analysis, as shown in Appendix A.1. Once we find the expression corresponding to each block, we can add instructions to the program to obtain a variable corresponding to each expression. The resulting program is as follows:

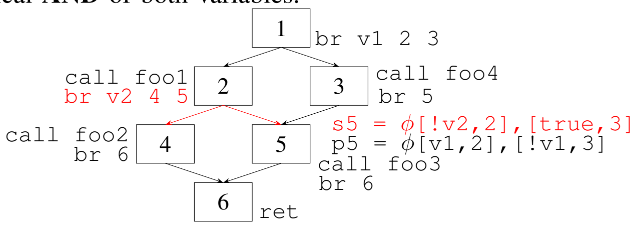


Naive method. The naive method discussed earlier, where the transformed program executes each block in the topological order, regardless of its condition, results in the following CFG:



However, this approach is not optimal. For instance, given that the variable v1 is public, we know that if v1 is false, blocks 2 and 4 do not need to be executed. Nonetheless, the method described above requires the execution of every basic block. Furthermore, there can be deep programs with one of the branch conditions being private; the current method leads to an avoidable exponential blowup. To evade this, instead of using a single condition variable per block, we maintain both a public and a private condition variable to manage block execution better.

Map {public, private} condition variables for each block. For optimal results, we associate two condition variables with each expression: one for public and one for private conditions. The final condition for a block is the logical **AND** of both variables.

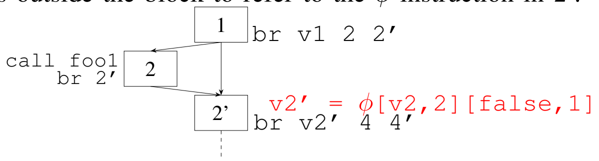


For instance, for block 5, where the condition is chosen from the set {(v1 & !v2), !v1}, we add two ϕ instructions (p5,s5) for public and private conditions. Using two separate ϕ instructions preserves correctness, as the ϕ instruction selects the value from the last executed block. Appendix A.1 describes how to find the condition expression with the corresponding variables for each basic block and how to add instructions to create a map for the corresponding conditions.

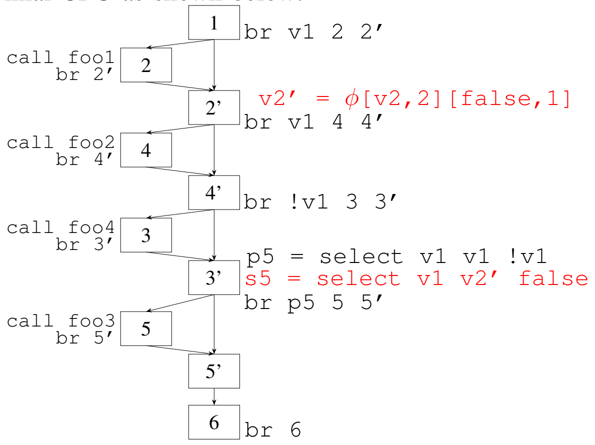
Update store and ϕ instructions. As discussed earlier, unnecessary execution of a basic block only impacts the program state due to ϕ and store instructions. Appendix A.3 shows how to update the store instructions such that they only impact the program state if the private condition corresponding to the basic block is true. We first load from the pointer and then use a mux to select based on the condition variable associated with the basic block between the value defined in this block and the value already written at the pointer. Finally, we store the output of mux.

Additionally, we replace all the ϕ instructions in the CFG with select instructions. For instance, in the final CFG of the example program, the instruction p5 = ϕ[v1,2],[!v1,3] is transformed into p5 = select v1 v1 !v1. It is important to note that a ϕ instruction selects a value based on the last executed block, not just checking which predecessor block was executed. Therefore, when converting ϕ instructions to a series of select instructions, we must account for the dominance relationships between predecessor blocks. Section 4.2 describes how to do this transformation correctly.

Update CFG. Instead of executing each basic block regardless of its execution condition, we execute it based on the associated public condition. Starting from the entry block, we branch to the next block in topological order, depending on its public condition. For the example program, we create a new empty basic block $2'$ and branch from block 1 to block 2 if v1 is true, or to block $2'$ if v1 is false. Now, in block $2'$, we insert ϕ instructions for any values defined in block 2 that are used outside the block 2 and replace the uses outside the block to refer to the ϕ instruction in $2'$.



This process is repeated for all blocks in topological order. Note that when we branch from $3'$ to 5, the branch instruction is `br p5 5 5'`. However, `p5` is defined in block 5. Thus, when replacing the ϕ instructions in block b to select instructions, we check if the ϕ instruction is the private or public condition variable for b. If yes, we move it before the terminator of the predecessor of parent block of b after the CFG has been updated to remove the private edges, i.e., `p5,s5` are moved from block 5 to block $3'$ in the final CFG as shown below:



Updating the terminators by the above-described method for loops can result in incorrect execution. Section 4.3 describes handling CFGs containing loops. Note that topological order is only defined for directed-acyclic graphs; thus, we show how to obtain a reverse post-order for CFGs with loops in Appendix A.2.

4.2. Update PHI (ϕ) Instructions

To replace a ϕ instruction with a sequence of `select` instructions, we first need to determine the order in which selections will occur based on the dominance relationships between the incoming basic blocks. We describe a step-by-step approach to the transformation as follows:

1) **Group incoming basic blocks:** We group all incoming basic blocks of the ϕ instruction according to their private condition and create a map where the key is the private condition, and the value is a list of all incoming blocks that share the same private condition.
2) **Order the blocks:** For each entry in the map, we arrange the basic blocks according to their post-order traversal. This ensures that the selections are processed in the correct order of execution.
3) **Select values for each private condition:** Once the blocks are ordered, we obtain values for each private condition as follows:
    a) We start by selecting the value from the first block in the list and use that as the initial value.
    b) For each subsequent block, first retrieve the value it passes into the ϕ instruction. Then, for each block, add a `select` instruction that chooses between this value and the previously selected value based on the public condition of the current block.
4) **Final value:** Once values have been assigned for each private condition in the map, we repeat the process for all private conditions using the order of the last blocks corresponding to each condition. This gives us a final selected value.
5) **Replace the ϕ instruction:** We replace all uses of the original ϕ instruction with the final value from the sequence of `select` instructions and remove the original ϕ instruction from the code.
6) **Maintain dominance from users:** We collect all the corresponding `select` instructions in a list for each basic block's ϕ instruction related to its condition variable. Once the control flow graph (CFG) has been updated, we move all these instructions to the predecessor of the parent block.

4.3. Update CFG

The entry block remains unchanged. We use lastBB to denote the last visited basic block in the CFG.

For simplicity, let us say that the CFG has no loops. When visiting a basic block b, there are two possible cases:

- **Public condition is constant true:** If the public condition of b is a constant true, we replace the terminator of the lastBB with an unconditional branch instruction that jumps to b, then set lastBB to b.
- **Public condition is variable:** If the public condition of b is a variable var, then we create an empty basic block b' and set the terminator of the lastBB to a branch instruction with condition var that branches to b when true and b' when false. Next, we clone the terminator of b as the terminator of b' and then set the terminator of b to unconditional branch instruction to b'. Then, for every instruction in b that is used outside this basic block, we add ϕ instruction to b' such that it has two incoming values, one from b and a default constant value from lastBB. Now, set lastBB to b'.

Now, consider there are loops in the CFG, and let us say we are visiting a basic block b, which is a loop latch (b has an edge to a block that dominates b). Note that when we visit this block, the terminator is maintained because we either do not change it or copy the terminator in b'. However, when we visit the next block in the order, the terminator of lastBB is lost. Thus, the loop no longer exists in the program. To address this, if lastBB is a loop latch, we create a new basic block lastBB' and replace the other successor (not loop header) to lastBB'. Further, when visiting b, if we create b' where b is a loop latch, then for the ϕ instructions in the corresponding loop header, we update the incoming block from b to b'.

Another issue with loops in the CFG is that if b is a loop header (first block in a loop) with a variable public condition, then the resulting program has a loop with two entry points, b, and b'. This renders the dominator relationships within the loop non-trivial to determine. Thus, instead of updating the terminator of b to b', we store b' as the new loop exit block corresponding to the header b and set lastBB = b. When we find that lastBB is a loop latch, we check if we have a new exit block for the corresponding loop header. If we find an exit block, then we use this as lastBB' instead of creating a new empty basic block. Since we have created a new loop exit, for every value defined in the loop and used outside the loop, we add ϕ instructions to the loop exit block, such that `v = `ϕ`[val,latch][default,loop preheader]`.

We defer the formal algorithm description and its proof of correctness to the Appendix A.4 and A.5, respectively.

5. Optimization Passes for MPC

After program transformation, we use the LLVM optimizer to perform all the optimizations other than vectorization passes. We write several custom optimization modules to ensure maximum vectorization of the operations that need MPC library functionalities. Below, we discuss the optimization functionalities and their implementations.

5.1. Vectorization

As discussed in section 3.2, secure computation protocols typically benefit from SIMD operations. Below, we describe in detail how we update the LLVM loop-vectorizer to suit MPC requirements, as well as the additional transformation passes we implement to fully exploit the SIMD opportunities in the program.

Recall from section 2.2 that the LLVM loop-vectorizer supports scalable-width instructions if the target supports them. However, the instructions that require secure computation always benefit from scalable-width instructions. Thus, we update the loop vectorizer to use scalable-width vector instructions for the loops with instructions handling private data. The vectorizer executes the plan with the lowest cost based on cleartext instruction costs. In MPC, vector instructions for non-linear operations cost significantly less than their scalar counterparts. Thus, we update the cost model of the loop vectorizer such that the non-linear operations that are to be computed using the MPC library have the minimum cost if they are vectorized using a scalable width and the maximum cost if they are scalar. These modifications are sufficient to vectorize simple loops like one that computes the sum of elements of two arrays. However, for more complex loops like cases where only partial vectorization is legal, the vectorizer avoids vectorization entirely.

Thus, we propose a solution to identify such patterns and split the loop into multiple parts so that the loop vectorizer can vectorize where advantageous. The first step is to flatten nested loops into loops with minimum possible depth without unrolling. Then, for the innermost loops, we identify the patterns that prevent their vectorization and split the loop into multiple parts so that some are vectorizable. Subsequently, we reconstruct the nested loop if the non-vectorizable parts are flattened loops. Lastly, we introduce a memory alignment pass since the LLVM vectorizer cannot vectorize operations involving non-contiguous memory. Before executing these passes, we use the LLVM utility to simplify the loops and transform them into LCSSA (loop closed SSA, where loop-defined values are only used within the loop, including uses in ϕ instructions in the loop exit block). Below, we discuss the more straightforward case,i.e., non-nested/innermost loop, and defer the loop flattening and reconstruction passes useful for nested loops to Appendix B.

Loop splitting. We focus exclusively on loops with a straight-line control flow, i.e., only one basic block. First, we use the LLVM memory dependence checker to detect any memory dependencies in the loop that prevent the vectorization of the loop. If dependencies are present, we apply the LLVM loop-distribute pass, which distributes the loop to resolve dependencies that inhibit vectorization whenever feasible. Thus, our transformation pass does not handle loops with unresolved memory dependencies.

A critical observation is that, since there is no memory dependence across iterations, the dependencies arise only from the ϕ instructions in the loop header. Thus, the first step is to categorize ϕ instructions into sets, such that the instructions within the same set do not depend on each other and order the sets according to the dependence. Using these ordered ϕ instruction sets, we identify the distinct sections (or parts) of the loop.

We create a part for each set that includes instructions dependent on prior sets but independent of the current set under consideration. Furthermore, if the set contains reduction ϕ instructions, we create a separate part to parallelize the private reduction operations. For any remaining ϕ instructions, we create an additional part with these instructions, along with any dependent instructions not already assigned to previous parts. We note whether the last part is vectorizable and contains private instructions to minimize unnecessary partitioning. If the first part for the current set contains no private instructions, we merge it into the last part. Next, we merge the last part corresponding to the set into the previously created part if it is not vectorizable.

Once we have a sequence of ordered parts, we identify the instructions used outside the part for each part. For each such instruction, we allocate space according to the instruction type and the number of loop iterations. We then create a new loop for each part and add store instructions for each value to the corresponding pointer used outside the new loop. We add a `load` instruction for any value used from the prior parts and replace the uses within the part with the loaded value. Finally, we deallocate each newly allocated pointer after its last use. Note that instructions that only require local computation can be duplicated instead of storing the intermediate results in memory.

Memory alignment. We utilize LLVM's loop vectorization legality analysis to detect any non-consecutive pointer accesses (either `load` or `store`) that may hinder loop vectorization. When such accesses are identified, we allocate memory based on the loop's iteration count for the relevant `load` and `store` instructions. For load instructions, we create a new loop preceding the original loop, move the identified `load` instructions to this new loop, and store the loaded values in the corresponding allocated space. In the original loop, replace each `load` instruction with a new one that accesses the allocated memory at the computed address (the base pointer plus the induction variable).

Similarly, we replace the `store` instructions in the original loop with `store` instructions at the corresponding pointer plus induction. We create a new loop following the original loop. We then move the non-consecutive `store` instructions to this post-loop, and the values to be stored are loaded from the corresponding pointer plus induction.

5.2. Parallelization of Reductions

Parallelization of reduction patterns is a standard optimization implemented and used in prior works [BHWK16], [Kel20]. We observe that the vectorization modules in LLVM can already detect reduction operations and replace them with intrinsic function calls like `llvm.vector.reduce.add.<type>`, these intrinsic functions are usually replaced by the target-specific optimal implementation for the respective reduction operation. Furthermore, the loop-splitting transformation described in Section 5.1 separates reduction operations from any non-vectorizable computation to enable the above-described LLVM optimization. Thus, to finally parallelize, we implement an interface for reduction operations that takes a pointer to the array, the size of the elements, and the opcode for the operation and uses a tree-like structure to compute the result. We replace the intrinsic function calls with function calls to this interface. Section 6 describes how to obtain the pointer to the input array given the input vector virtual register and how this interface is implemented and linked to the output program.

5.3. Loop Coalescing

ObliVM [LWN+15] introduces Java-like DSL and proposes an optimization technique that transforms a nested loop with depth two into a single loop with an overall iteration count $O(n{+}m)$ instead of $O(n{*}m)$, where n, m are the maximum iteration counts for the outer and inner loops respectively. We implement their technique for LLVM-IR, which involves handling the ϕ instructions in the headers of the loops, maintaining the dominance relationship between the instructions and their uses after loop coalescing and cases where the loop latch is not the block that exits the loop. We use the LLVM metadata `unroll_count` to get the max loop count and the condition analysis described in Appendix A.1 to find if the loop has a private latch. Additionally, we use the LLVM loop analysis to determine the components of a loop, i.e., preheader, header, latch, and exit blocks. This helps us implement the transformation in $\approx$ 500 lines of code. The transformation algorithm is described in Appendix C.

6. End-to-End Integration

As discussed earlier, the input program is essentially the same as that for plaintext computation, and the user provides an input JSON file to our compiler to identify the nature of the input parameters and output value for each function to be analyzed. An example program and the corresponding JSON are shown in Figures 4 and 5, respectively. First, we use a general-purpose compiler (e.g., Clang) to obtain the corresponding LLVM-IR and run some

```
void histogram(int *A, int *B, int length,
        int *ret, int bins) {
  memset(ret, 0, bins * sizeof(int));
  for (int j = 0; j < bins; ++j)
    for (int i = 0; i < length; ++i)
      if (A[i] == j)
        ret[j] += B[i];
}
```

Figure 4: The code for histogram, same as the cleartext program.

```
{"histogram": {
    "input": [priv, priv, pub, priv, pub],
    "output": pub
}}
```

Figure 5: **JSON specification.** Specification for private (`priv`) and public (`pub`) parameters of function described in Figure 4.

non-aggressive LLVM optimizations (e.g., all optimizations in the O1 pipeline). Then, we use the JSON and the LLVM-IR to analyze the program and identify whether each instruction depends on a private value. Subsequently, we apply the LLVM optimizer, and our custom transformation passes to produce an optimized and oblivious version of the original program in LLVM IR. Finally, we transform the IR by replacing the instructions that require secure computation with function calls to the MPC library and adding declarations for the respective functions to the IR. The definitions of these functions need not be known at the time of transformation. When compiling the IR to an object file, one needs to add linker flags to the library that implements the MPC interface.

6.1. Handle Instructions for Private Values

As we know, after the analysis of the program given the JSON, each instruction is labeled private if any of its input variables are private. For all the following discussions, we assume that the MPC backend uses the same data type for private values as cleartext computing and defer the use of custom MPC data types to Appendix D. We do not need to replace instructions that are linear operations on the private values depending on the type of sharing. However, non-linear operations like `mul` of two private values must be replaced with the corresponding MPC function call. Let us assume we have the following instruction:

```
%c = mul i32 %a, %b
```

then we use a corresponding MPC function interface for `mul` over `i32` data type:

```
int32_t mulI32(int32_t a, int32_t b)
```

After identifying the instruction, we add a declaration for the corresponding function to the program if it doesn't exist already and then replace the instruction as follows:

```
%c = call i32 @mulI32(i32 %a, i32 %b)
```

Vector instructions of scalable width. Let us start with an example of a vector instruction with scalable width:

```
%c = mul <vscale x 1 x i32> a, b
```

where `vscale` depends on the target architecture and `a` and `b` are previously defined vector virtual registers. The corresponding function for the MPC interface takes pointers for the input pointers, the output pointer, and the number of values, for instance:

```
void mul(int32_t *a, int32_t *b, int32_t *c,
    int n)
```
such that for $i \in [0, n-1]$: $c[i] = a[i] * b[i]$.

The first step is to determine the pointers for the output and input registers to pass as function arguments. If the output instruction is used within a vector `store` instruction at the pointer `c.ptr` (a scalar value), then we use `c.ptr`; otherwise, we allocate space for `c`, and free it after all uses of `c`. After finalizing the output pointer, we store it in a map to replace references to `c` in its uses later. Similarly, if the instruction corresponding to the input virtual register is a `load` instruction from the pointer `a.ptr`, we use `a.ptr`; however, if it points to another instruction, we must retrieve the corresponding pointer for the input instruction from the instruction-to-pointer map.

Another challenge we have yet to discuss is obtaining `n`, as `vscale` is a target-dependent value and may not be supported on all targets. Ideally, we want `n` to be the original loop count (i.e., the number of times the non-vectorized loop executes). When vectorizing a loop using scalable-width vector instructions, the loop vectorizer transforms the original loop into two loops: the first loop (vectorized loop) iterates `n // vscale` times, and the scalar loop iterates `n % vscale` times, assuming the maximum possible vector width is `vscale`. Additionally, metadata is added to the loop to indicate that it has been vectorized without requiring any new analysis. We use this pattern to identify the corresponding value `n`, ensuring the scalar loop becomes unreachable once we replace all vector instructions with MPC function calls.

Linking to the MPC library. As discussed earlier, Smaug only knows the mapping of instruction code, data type and the corresponding function names. The user can implement the MPC interface functions using their choice of library and protocols. We implement the GC-based interface using EMP-toolkit [WMK16] (built as `libgc.so`) and an interface using a GMW-based circuit evaluation [GYKW24] and the circuits generated by EMP-toolkit to evaluate primary operations,e.g., addition, multiplication (as `libgmw.so`). When using the code-generator to build an object file, we pass a linker flag to the library with the interface we want to use, `-lgc` (resp. `-lgmw`) for the GC (resp. GMW) interface.

7. Evaluation

We summarize the key findings from our evaluation below:

1) **Performance of Compilation Phase:** We find that Smaug outperforms COMBINE, CBMC-GC, and MP-SPDZ by orders of magnitude and has a performance comparable to Obliv-C.
2) **Performance of Result Programs:** The resulting programs either outperform or perform comparably to those implemented using other compilers for both GC and GMW-based interfaces.
3) **Adaptability for multiple languages:** Smaug can compile programs written in languages that support LLVM-IR, including C/C++ and Rust.

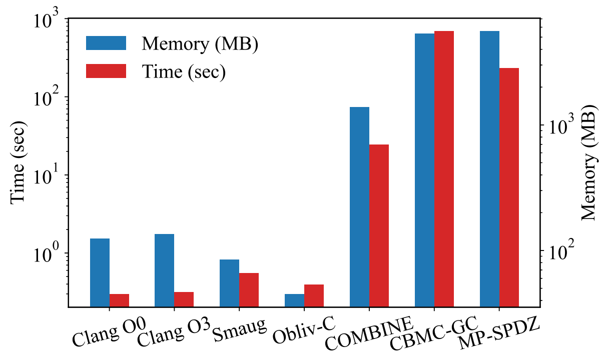


Figure 6: **Comparison of average time and memory usage.** Note that we are unable to compile some programs with CBMC-GC, and thus, they are omitted from their average.

4) **Real-world programs:** We compile several open-source programs with minimal changes to enable secure computation.

All experiments are conducted on AWS instance type `c5.2xlarge`. We use the standard benchmark suite from [LIS+23] to compare with prior works. It is described in Appendix E.1. For prior works, we use COMBINE [LIS+23], CBMC-GC-2 [HFKV12] from the **BYOC** group of compilers and Obliv-C [ZE15], MP-SPDZ [Kel20] from the **BYOL** group. Note that [YPHK23] achieves the state-of-the-art performance in the **BYOM** group; however, they do not provide a tool to compile a high-level program into their ISA.

7.1. Performance of the Compilation Phase

We benchmark the time and average memory required to compile the high-level programs into an executable by the prior works and Smaug. Figure 6, shows a comparison of the average time and memory used by the compilers for compiling all the programs in the benchmark suite. We note that CBMC-GC is unable to compile several programs as it generates a circuit during compile time, thus requiring significant RAM usage. We observe that Obliv-C performs slightly better than Smaug; however, it does not optimize the program and only supports garbled-circuits.

Compilation Time. Figure 6 shows that the compilation time for Smaug is significantly better than COMBINE, CBMC-GC, and MP-SPDZ (this order of prior works is used for all the following comparisons). On average, Smaug performs $44\times$, $1240\times$, and $420\times$ better than the prior works. Obliv-C and Smaug have comparable performance, where Obliv-C slightly outperforms Smaug by a factor of $1.4\times$. Table 6 shows a detailed comparison of the time taken to compile each program in the benchmark suite. For instance, when compiling the histogram benchmark (Figure 4), Smaug outperforms prior works by a factor of $48\times$, $1293\times$, and $187\times$.

Peak Memory Used by the Compiler. Figure 6 shows a similar conclusion for the peak memory usage averaged over all the benchmarks, where Smaug outperforms COMBINE, CBMC-GC, and MP-SPDZ and has a comparable performance to Obliv-C. Smaug uses $16\times$, $63\times$, and $65\times$

Program	GMW								GC	
	Circuit Size				Depth				Circuit Size	
	Smaug	COMBINE	CBMC-GC	MP-SPDZ	Smaug	COMBINE	CBMC-GC	MP-SPDZ	Smaug	Obliv-C
Biometric	17.8M	23.9M	-	24.6M	270K	753K	-	1.26M	18.3M	17.9M
Convex Hull	4.35M	16.0M	11.8M	10.5M	389	2.51K	263	1.97M	4.35M	4.41M
Count10*	393K	1.04M	734K	802K	12.7K	1.32M	8.27K	16.4K	401K	1.03M
Count10*2	520K	10.4M	755K	802K	12.7K	1.32M	8.27K	16.4K	532K	647K
DB Variance	4.44M	6.53M	-	5.76M	2.10K	2.62M	-	16.7K	4.57M	4.44M
Histogram	3.21M	3.91M	4.11M	3.31M	2.33K	1.33M	4.18K	614K	3.23M	-
Inner Product	4.19M	5.23M	-	5.00M	403	1.31M	-	8.43K	4.32M	4.19M
k-Means	4.98M	6.81M	5.06M	7.63M	191K	261K	983	239K	5.11M	157M
Longest10*2	1.04M	1.42M	1.13M	1.32M	286K	2.06M	28.6K	65.5K	1.06M	1.04M
Max Dist.	778K	1.15M	1.20M	1.05M	270K	2.03M	32.7K	65.5K	782K	778K
Minimal Points	274K	737K	729K	528K	131	872	70	114K	274K	274K
MNIST ReLU	262K	368K	-	294K	33	12.4K	-	15	262K	131K
PSI	33.5M	33.5M	-	66.0M	1.05K	4.16K	-	12.5M	34.6M	33.5M

TABLE 1: **Circuit size and depth comparison.** CBMC-GC cannot compile the missing benchmarks due to high memory usage. Histogram compiled with Obliv-C does not terminate for the input size used.

	Compiler	512	1024	4096
Compile Time (s)	Smaug	0.53	0.53	0.53
	COMBINE	28.35	27.98	27.20
	MP-SPDZ	13.2	27.29	104.81
	CBMC-GC	628.69	644.17	741.00
	Obliv-C	0.87	0.87	0.87
Compilation Memory	Smaug	81.51	81.51	81.51
	COMBINE	1,408.43	1,408.43	1,323.96
	MP-SPDZ	407.92	246.48	1,795.37
	CBMC-GC	3,857.13	4,216.55	8,063.34
	Obliv-C	43.17	43.17	43.17
Run-time	Smaug (GMW)	0.15	0.3	1.25
	COMBINE	155.34	316.01	1292.59
	MP-SPDZ	1.14	2.27	8.96
	Smaug (GC)	0.23	0.34	2.44
	Obliv-C	0.25	-	-

TABLE 2: **Comparison with prior works w.r.t. input size for Histogram with** 5 **bars.** The Obliv-C benchmark does not terminate for input sizes 1024 and 4096.

less memory when compared to the prior works. Table 7 shows a detailed comparison of the peak memory used by the compiler for each program. Prior works require $16\times$, $100\times$, and $37\times$ more memory than Smaug for compiling the histogram benchmark.

Comparison for different input sizes. Table 2 compares Smaug with prior works, varying the size of the dataset input to compute a histogram with 5 bars. It can be seen that for MP-SPDZ and CBMC-GC, the computational resources required for compiling a program increase with the input size, while for others, it is independent of the input size.

7.2. Performance of the Result Programs

We compare the circuit size and depth of Smaug's output programs with those of all the prior works. Furthermore, we compare the execution time of the compiled programs. We use a GC-based interface when comparing with Obliv-C and a GMW-based interface when comparing with COMBINE, CBMC-GC and MP-SPDZ. For a fair comparison, we use MOTION [BDST22] as the backend for COMBINE [LIS+23] because it does not perform any additional optimizations.

Circuit Statistics Comparison. Table 1, shows a detailed

Program	No Vector-ization	Split. + Vect.	Flatten + Split. + Vect.	Flatten + Split. + Recon. + Vect.
Biometric	1794048	778240	778302	270522
Histogram	1945600	2180	635007	127103
Minimal Points	270464	8384	4163	131

TABLE 3: **Circuit depth comparison with respect to our transformation passes.**

comparison of circuit size and depth. For the garbled-circuits-based backend, we can see that the number of gates is similar to that of Obliv-C and sometimes even better ($3\times$ smaller for k-Means). In terms of circuit size for GMW, Smaug outperforms the other compilers or has a comparable circuit size. The depth of the circuit is significantly better than both COMBINE and MP-SPDZ; namely, for the histogram benchmark when compiled with Smaug, the circuit has $570\times$ (, resp. $263\times$) lower depth in comparison to COMBINE (resp. MP-SPDZ). CBMC-GC optimizes the circuit at low-level, thus the resulting circuits have lower depth than those of Smaug; however, for most cases, the difference is not as significant, and the resources used during compilation are orders of magnitude higher than Smaug.

Runtime Comparison. Figure 7 compares the execution time of all programs in the benchmark suite compiled by Smaug, COMBINE, MP-SPDZ, and Obliv-C. We observe that the programs with GMW-based backend compiled with Smaug consistently outperform those compiled with COMBINE and, in most cases, have comparable performance to those compiled with MP-SPDZ. For instance, for biometric matching, Smaug results in a program that is $157\times$ faster than COMBINE and $2\times$ faster than MP-SPDZ. The programs with a GC-based backend compiled with Smaug outperform those compiled using Obliv-C, ranging from $1.5\times$ for convex hull to $250\times$ for k-Means.

Ablation Study. Table 3 shows how the depth of the circuit changes with the transformation passes used. For biometric and minimal points we see that the program is optimal when all three transformation passes, i.e., loop splitting, loop flattening, and loop reconstruction are used. We note that histogram does not see this improvement, and instead, the

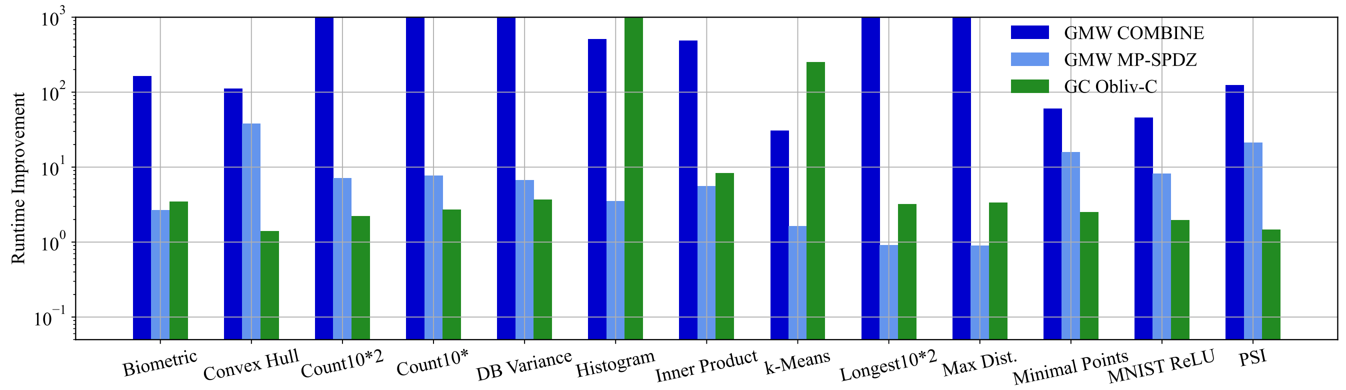


Figure 7: **Runtime improvement for programs compiled by COMBINE, MP-SPDZ, and Obliv-C vs Smaug.** DB-Variance, Longest10*2, and maximum distance compiled by COMBINE are killed before the process finishes. Histogram does not terminate when compiled with Obliv-C for the input size.

Program	Rust		C++	
	Depth	Size	Depth	Size
Histogram	657220	1924965	2180	2580325
MNIST Relu	262144	33	262144	33

TABLE 4: **Rust vs C++ benchmarks.** Comparison of circuit size and depth for the programs when written in Rust and C++.

performance worsens because of the choice to prefer vectorization over parallelization in the loop reconstruction pass. As shown in Figure 4, `bins` is much smaller than the size of the dataset (`n`), thus reduction is beneficial in comparison to `n` SIMD operations. Hence, we see that the loop flattening and reconstruction passes might be counterproductive when the inner loop has reduction operations, and the outer loop count is smaller than the inner loop count.

7.3. Adaptability for Multiple Languages

Smaug can compile programs written in any language that can be compiled into LLVM-IR. All the prior benchmarks use C++ implementation of the programs; thus, we compile a subset of the benchmarks in Rust, to demonstrate the frontend-agnostic property. Below is an example of Rust implementation for histogram:

```
fn histogram(a: &[i32], b: &[i32], ret: &mut
   [i32]) {
   let n = a.len();
   let d = ret.len();
   for j in 0..d {
      ret[j] = 0;
      for i in 0..n
         if a[i] == j as i32
            ret[j] += b[i];
   }
}
```

Table 4 shows a comparison of the circuit size and depth for programs using Rust and C++. We observe that the circuit size and depth can sometimes differ for Rust programs because of the additional checks involved or other differences in the frontend of both languages. We note that rust programs include panics; however, we need the CFG to have only `br` and `ret` instructions for our transformation to the non-oblivious algorithm. Thus, if the CFG for a function is non-oblivious, we first remove the panics from the function. In future engineering efforts, we aim to extend the transformation algorithm to support all the terminators in LLVM-IR.

Program	GC-time (s)	GMW-time (s)
kmeans	23	28
Minesweeper	0.19	0.21
Blackjack	0.007	0.003

TABLE 5: **Real-world programs.** For k-Means, the time shown to run it on a dataset of length 150 with 2 features and 3 clusters. For Minesweeper, the time shown is the average response time after each move when played over an 8×8 grid.

7.4. Real-World Programs

We use Smaug to build several real-world open-sourced programs with minimal changes to enable secure computation. We test the following three programs and show their performance in Table 5:

- **K-Means**[1]**:** We build an open-source K-Means implementation with minimal changes. We use this library to classify secret-shared data among two parties in a MySQL database and use the standard MySQL library in C++ to get the input data.
- **Minesweeper**[2]**:** We first update a public Minesweeper implementation such that the location of mines is known to the dealer and the player's move remains hidden from the dealer. We build it using Smaug to enable secure computation and benchmark the average time taken by the player to obtain the board after making its move.
- **Blackjack**[3]**:** This is a single-player blackjack game where the computer is the dealer and the player bets on getting a certain combination of cards. On each turn, the player can choose between two moves. We update this game so that randomness is shared among two parties, one being the dealer and the other being the player. We then build it using Smaug and benchmark the average response time after each move.

1. https://github.com/KlimentLagrangiewicz/k-means-in-C
2. https://www.geeksforgeeks.org/cpp-implementation-minesweeper-game
3. https://github.com/ineshbose/Blackjack_CPP

8. Conclusion

In this work, we showed that LLVM can be extended to incorporate most MPC optimizations and achieve performance competitive to, often better than, prior tools relying on custom languages and/or compilers. This is a new way toward maintainable and reusable cryptography compilers that can potentially be applicable to many other protocols, including zero-knowledge proofs and fully homomorphic encryptions. Future works also include extending Smaug to support secure RAM accesses and mix-mode operations.

Acknowledgements

This work is supported by NSF awards #2236819 and #2318975.

References

[All70] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 1970. 2.1

[ARG+21] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C. Myers, and Elaine Shi. Viaduct: an extensible, optimizing compiler for secure distributed programs. In *PLDI*, 2021. 3

[BDK+18] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In *ACM CCS*, 2018. 2

[BDST22] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. Motion – a framework for mixed-protocol multi-party computation. *ACM Trans. Priv. Secur.*, 2022. 1, 7.2

[BHK+23] Lennart Braun, Moritz Huppert, Nora Khayata, Thomas Schneider, and Oleksandr Tkachenko. FUSE - flexible file format and intermediate representation for secure multi-party computation. In *ASIACCS*, 2023. 3.3

[BHWK16] Niklas Büscher, Andreas Holzer, Alina Weber, and Stefan Katzenbeisser. Compiling low depth circuits for practical secure computation. In *ESORICS*, 2016. 1.2, 3.2, 5.2

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *ACM TOPLAS*, 1991. 2.2

[CGR+19] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. Ezpc: Programmable and efficient secure two-party computation for machine learning. In *IEEE EuroS&P*, 2019. 3, 3.1

[CZO+23] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Wenting Zheng. Silph: A framework for scalable and accurate generation of hybrid MPC protocols. In *IEEE S&P*, 2023. 2

[GF95] Anwar M. Ghuloum and Allan L. Fisher. Flattening and parallelizing irregular, recurrent loop nests. In *PPOPP*, 1995. 1.2, 3.2

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *ACM CCS*, 2012. 2.3

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *ACM STOC*, 1987. 3.2

[GYKW24] Radhika Garg, Kang Yang, Jonathan Katz, and Xiao Wang. Scalable Mixed-Mode MPC. In *IEEE S&P*, 2024. 6.1

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011. 1

[HFKV12] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In *ACM CCS*, 2012. 2, 1.2, 3.1, 3.2, 7

[HST+21] Tim Heldmann, Thomas Schneider, Oleksandr Tkachenko, Christian Weinert, and Hossein Yalame. LLVM-based circuit compilation for practical secure computation. In *ACNS*, 2021. 2, 1, 3.1

[Kel19] Marcel Keller. The oblivious machine - or: How to put the C into MPC. In *Latincrypt*, 2019. 1, 1

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *ACM CCS*, 2020. 3, 1.2, 5.2, 7

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE CGO*, 2004. 2.2

[LIS+23] Benjamin Levy, Muhammad Ishaq, Benjamin Sherman, Lindsey Kennard, Ana L. Milanova, and Vassilis Zikas. COMBINE: COMpilation and backend-INdependent vEctorization for multi-party computation. In *ACM CCS*, 2023. 2, 1, 1.2, 3.1, 3.2, 7, 7.2

[LWN+15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *IEEE S&P*, 2015. 3, 3.1, 3.2, 5.3

[MGC+16] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin R. B. Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. *IEEE EuroS&P*, 2016. 3, 3.1

[RLT15] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, 2015. 2.3

[SHS+15] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *IEEE S&P*, 2015. 1

[SZD+16] Ebrahim M. Songhori, Shaza Zeitouni, Ghada Dessouky, Thomas Schneider, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. Garbledcpu: a mips processor for secure computation in hardware. In *ACM DAC*, 2016. 1, 3.1

[SZE+16] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. TaoStore: Overcoming asynchronicity in oblivious data storage. In *IEEE S&P*, 2016. 2.3

[TSS+17] Xinmin Tian, Hideki Saito, Ernesto Su, Jin Lin, Satish Guggilla, Diego Caballero, Matt Masten, Andrew Savonichev, Michael Rice, Elena Demikhovsky, et al. LLVM compiler implementation for explicit parallelization and SIMD vectorization. In *LLVM-HPC*, 2017. 2.2

[WGMK16] Xiao Shaun Wang, S. Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure computation of MIPS machine code. In *ESORICS*, 2016. 1, 3.1

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016. 1, 6.1

[YD23] Qianchuan Ye and Benjamin Delaware. Taype: A policy-agnostic language for oblivious computation. In *PLDI*, 2023. 3

[YPHK23] Yibin Yang, Stanislav Peceny, David Heath, and Vladimir Kolesnikov. Towards generic MPC compilers via variable instruction set architectures (VISAs). In *ACM CCS*, 2023. 1, 3.1, 7

[ZE15] Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. https://eprint.iacr.org/2015/1153. 3, 1.2, 3.1, 7

[ZSB13] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: a general-purpose compiler for private distributed computation. In *ACM CCS*, 2013. 3, 3.1

Appendix A. Transformation to Oblivious Programs

A.1. Find Condition for Basic Block

We first find an expression corresponding to the execution condition of each basic block. For the entry block, the condition is always true. The execution condition of a basic block is determined by the combination of the condition expressions and branch conditions of its predecessors (not dominated by the block), along with the conditions for all its direct descendants in the post-dominator tree. Using the post-dominator analysis helps simplify and minimize the overall condition expression. For the example program, the condition for execution of block 6 obtained without the post-dominator analysis is the condition for block 4 or 5. Block 6 post-dominates 1, 2, 3, 4, and 5, and we know that block 1 dominates 2, 3, 4, and 5. Thus, the condition expression for block 6 is the same as that of block 1. This way we can eliminate the predecessor blocks and simplify the condition.

Given the condition expression corresponding to each basic block, we insert instructions in the IR to obtain variables for the execution condition. We keep track of two variables, one for private and another for public conditions, such that the expression evaluates to the logical and of both variables. The expression can be viewed as a tree, where and nodes have two children, and or nodes (i.e., sets) have more than one. For and nodes in the tree, we add two `and` instructions to get the corresponding private and public condition variables. The private (public respectively) condition variable is the and of the private (public respectively) condition variables of the left and right child. Similarly, for condition sets, we insert ϕ instructions for private and public condition variables. These ϕ instructions are eventually transformed into a series of `select` instructions as described in Section 4.2 and moved to the predecessor block.

A.2. Post Order

We first identify all the strongly connected components (SCCs) of the CFG in the post order using Tarjan's DFS algorithm. If an SCC has more than one basic block, we find the loop defined by the header as the last node in the SCC. For each loop, we consider the subgraph for the innermost loop without the latches (back-edges). If this is an innermost loop, we get a DAG. We use this DAG to obtain the post order for nodes in the SCC. If this is not the innermost loop, we consider the subgraph for the outermost loop without the edges and use the same method until we obtain an order for the nodes in this SCC.

A.3. Handle Store Instructions

The `store` instructions in any basic block with a variable private condition should only affect the program state if the variable evaluates to true at runtime. We use a sequence of `load`, `select`, and `store` instructions for every such store instruction. For a store instruction `store v, ptr` in a basic block with private condition `c`, we use the sequence of instructions:

```
%x = load <type> %ptr
%sel = select i1 %c, %x, %v
store <type> %sel, %ptr
```

A.4. Update CFG

Algorithm 1 describes how to update a CFG after we update the ϕ and `store` instructions.

Algorithm 1 Update CFG

```
 1: order: vector of basic blocks in F in reverse post order
 2: lastBB = order[0]
 3: newLoopExits: map of <loop header, new loop exit>
 4: oldNewBB: map of <BB', BB>
 5: pub: map of <BB, public condition var>
 6: priv: map of <BB, private condition var>
 7: for BB in order[1:] do
 8:     if lastBB is loop latch then
 9:         LH = Loop header corresponding to lastBB
10:         if newLoopExits[LH] then
11:             exit = newLoopExits[LH]
12:         else
13:             exit = new empty basic block
14:         update the other successor with exit
15:         if oldNewBB[lastBB] then
16:             lastBB: replace successors φ uses with oldNewBB[lastBB]
17:         lastBB = exit
18:     if pub[BB] = true then
19:         update terminator of lastBB to br BB
20:     else
21:         create empty basic block: BB'
22:         update terminator to lastBB to br pub[BB] BB BB'
23:         if BB is loop header then
24:             newLoopExits[BB] = BB'
25:         else
26:             copy the terminator of BB at the end of BB'
27:             set terminator of BB to br BB'
28:             store BB', BB in oldNewMap
29:             for instruction I: BB do
30:                 if I is used outside BB then
31:                     create φ for I in BB' with default value from lastBB and I from BB: I'
32:                     replace all uses of I outside BB and in terminator of BB with I'
```

A.5. Correctness

Our transformation algorithm assumes that the basic blocks of the input program have a trivial dominance relationship, the terminator instructions are either `br` or `ret`, and there is only one entry and exit block in the CFG.

Here, we prove the correctness of our transformation using a forward simulation technique where we say that at every point the state of the original program (S) and the new program (S_t) follow a relationship R, and the output of both the programs is the same. A state $S = (p, m, \mathbf{r})$ where p is the program counter, m is the memory and $\mathbf{r}$ is the set of all virtual registers. We say that the relationship $R(S, S_t)$ holds if:

- p_t belongs to a block in P_t that corresponds to a reachable state in P subject to a pair of condition flags f^s (private) and f^p (public).
- $m = m_t$.
- $\forall r \in \mathbf{r}$ we have a corresponding $r_t \in \mathbf{r_t}$, such that if the value of r is defined then $r = r_t$.

The entry node for both programs is the same; thus, the initial states of the programs P and P' are the same.

Theorem 1. *For each instruction I where $S \xrightarrow{I} S^n$ and $R(S, S_t)$ holds, $\exists I_t$ s.t. $S_t \xrightarrow{I_t} S_t^n$ and $R(S^n, S_t^n)$ holds.*

Proof. We consider the following cases of I as follows: **Basic Instructions:** We first discuss non-terminator instructions.

- `store` **instruction**: In P, consider a `store` instruction with operands `v`, `ptr`; this corresponds to a sequence of `load`, `select`, and `store` instructions denoted by I_t^0, I_t^1, and I_t. `v'` and `ptr'` are the registers corresponding to `v` and `ptr` in $\mathbf{r}'$.
  The `select` instruction chooses `v'` if f^s for the block in P is true; otherwise, it selects the old value at `ptr'`. Thus, I_t updates m_t conditionally based on f^s. Furthermore, I_t is reached in P' conditioned on f^p. Therefore, $R(S, S_t)$ is maintained.
- ϕ **instruction**: A ϕ instruction in P is transformed into a series of `select` instructions in P'. Let $\mathbf{b}$ represent the set of incoming blocks, and the value chosen comes from the last executed block, b_l. In P', we choose the value from a block in $\mathbf{b}$ that has both f^s and f^p true, and it comes last in the reverse post order of the CFG among all other blocks where both flags are true.
  Since b_l is last executed, it must follow all other immediate predecessors executed in P in the reverse post order. P' selects the correct value maintaining $R(S, S_t)$.
- **All other instruction types**: For all other instructions the main difference between I in P and I_t in P_t is the input registers. Since the operand registers maintain $r = r_t$, the relationship $R(S, S_t)$ holds for these instructions.

Control-Flow Instructions: For non-loop latch `br` instructions, the values defined in the current block b are propagated through ϕ instructions in a new block which is always executed in P'. We branch to the successor conditioned on the corresponding f^p. The relationship holds because a block in P' is only reached if it is possible to reach it in P, and new registers in P' are defined to correspond to the values defined in b but used outside.

We do not update the terminator if a loop latch does not equal the loop exit. Thus, $R(S, S_t)$ holds; otherwise, we replace the successor from the old exit block with a new block we created in P' for value propagation for the values that are defined in the loop but are used outside. This new exit block in P' branches to the old exit block. Thus, $R(S, S_t)$ holds for loop-latches and non loop-latches. □

From the above proof, we can see the relationship R holds for each instruction in P. Only new registers are introduced for all the additional instructions in P'; thus, the relationship is maintained.

A.6. Handle Private Loop Bounds

First, we run the loop-coalescing pass described in Appendix C. Next, for any loop with a private exit condition, there should be a pragma unroll loop_count, i.e., metadata information for the maximum number of iterations for the loop. We create a new header basic block and a latch block. We add a new induction variable in the new header, which is incremented in each iteration (in the new latch), and we take the back edge if the incremented induction does not equal the unroll_count. Furthermore, we add a ϕ instruction to the new header corresponding to the original loop exit condition, and we go to the original header if the exit condition is false; otherwise, we go to the new latch. We replace the latch and exit instructions from the original loop with unconditional branch instructions to the new latch. Finally, we run the transformation to oblivious CFG, given that no loops have private exit conditions.

Appendix B. Vectorization

Here we describe the loop flattening and reconstruction passes below:

Flatten nested loops. To flatten nested loops, we work from the outermost loop. If the skeleton of the loop L is b1 $\rightarrow$ l1 $\rightarrow$ b2 $\rightarrow$ l2 $\rightarrow$ b3 $\rightarrow$ b1, where l1 and l2 are sub-loops, then we create five new loops. The loops containing b1, b2, and b3 iterate for the same number as L. For the loops with l1 and l2, instead of creating new loops, we alter them. For instance, if l1 iterates m times and L n times, then the transformed loop l1 iterates n∗m times; thus, the depth of the resulting loops is strictly less than that of L. Note that we should not create new loops for the parts that are loops themselves as this allows us to reuse the existing loop analysis. In this optimization module, we restrict the outermost loop to have a straight control flow if the identified sub-loops are treated as a single node.

The first step is identifying the parts to make using the loop access analysis. If the part itself is not a loop, create a new loop; otherwise, update the inner loop. To update the inner loop, we update the latch condition instruction to compare with the new loop count (n∗m) and obtain the effective induction value for the previous outer loop (n) and inner loop (m) using division and unsigned remainder, respectively. These replace the references (except in the latch condition) with the respective induction variables with the computed values.

Program	Clang -O0	Clang -O3	Smaug	COMBINE	CBMC-GC	Obliv-C	MP-SPDZ
Biometric	0.315	0.34	0.586	31.21	-	0.385	127.426
Convex Hull	0.313	0.327	0.574	28.265	894.55	0.392	19.741
Count10*2	0.314	0.327	0.56	27.502	621.01	0.369	19.393
Count10*	0.314	0.329	0.561	27.485	619.99	0.377	407.047
DB Variance	0.314	0.327	0.547	-	-	0.371	345.321
Histogram	0.314	0.326	0.573	27.586	742.61	0.379	107.583
Inner Product	0.313	0.327	0.547	27.203	-	0.373	324.694
k-Means	0.105	0.131	0.405	28.28	744.62	0.413	397.522
Longest10*2	0.314	0.327	0.574	29.662	631.74	0.375	548.735
Max Dist.	0.314	0.33	0.58	28.883	632.89	0.368	504.992
Minimal Points	0.316	0.328	0.573	30.218	616.18	0.384	4.147
MNIST ReLU	0.313	0.327	0.546	28.493	-	0.363	156.1175
PSI	0.313	0.326	0.571	28.858	-	0.376	47.233

TABLE 6: **Comparison of compilation time (sec).**

Program	Clang -O0	Clang -O3	Smaug	COMBINE	CBMC-GC	Obliv-C	MP-SPDZ
Biometric	121.586	133.57	90.508	1,329.29	-	43.226	3,354.074
Convex Hull	121.742	125.52	80.324	1,352.23	16,090.59	43.843	411.766
Count10*2	121.883	133.57	80.285	1,322.89	1,482.01	42.441	829.531
Count10*	121.793	133.559	82.055	1,323.41	1,378.95	43.601	9,429.929
DB Variance	121.57	125.75	80.523	-	-	42.406	7,926.984
Histogram	121.688	133.195	80.84	1,323.96	8,143.34	43.011	3,023.008
Inner Product	121.688	125.934	80.387	1,313.29	-	42.574	7,880.918
k-Means	102.598	118.836	99.395	1,377.71	10,161.11	45.324	9,471.820
Longest10*2	121.793	126.031	96.961	1,325.66	1,941.31	43.062	13,470.066
Max Dist.	122.082	125.867	88.719	1,317.47	2,362.71	42.664	11,748.093
Minimal Points	121.645	125.57	81.055	1,337.49	1,137.85	43.664	107.094
MNIST ReLU	121.133	133.605	81.164	1,320.35	-	42.339	3,598.015
PSI	121.824	125.664	81.02	1,324.61	-	42.832	1,076.746

TABLE 7: **Comparison of peak memory(MiB) used during compilation.**

For each part, identify the values used in the part but defined outside the part and within L, except the induction variable. If any of the values used are a ϕ instruction with an incoming value from basic blocks in the following parts, then keeping the order of the parts the same, we combine the parts, i.e., do not split the loop until that value is contained with the original part. Note that if the part is an inner loop, then the incoming value in the ϕ instruction in the original loop header can itself be a ϕ instruction in the loop exit block with only one incoming value because of LCSSA representation. Similarly, for any load instructions in a part, check if any load instruction in the part has a read-after-write dependency with write (store instruction) in the parts that follow. In case of such a dependence, combine the parts until the dependence is within a single part.

Once we have identified the parts to form, we make separate loops, as discussed earlier. Next, identify the values defined in each part but are used outside this part and within L. For each such value, allocate memory in the corresponding loop preheader (block just before the header) and store the value at the corresponding index in each iteration. For all uses of this value in the parts that follow load from the same index. If this value is used in the ϕ instruction in the loop header, load from induction minus one if possible; otherwise, use the default value (coming for loop preheader).

When updating an inner loop, we change the loop count; thus, we need to consider the ϕ instructions being moved here from the outer loop and the existing ϕ instructions in the inner loop. For the existing ϕ instructions, we add a select instruction where if the inner induction (`urem` value) is zero, we choose the incoming value from the preheader and otherwise use the result of ϕ. Similarly, for ϕ instructions from the outer loop, we add a select instruction that chooses the new value, i.e., the ϕ value if the inner induction is zero and the value from the previous iteration otherwise. Finally, we add metadata `llvm.loop.flattened` so that the loop-reconstruction pass can identify the change.

Loop reconstruction. Next, for any loops that were flattened and are not vectorizable, but not just because of non-contiguous memory accesses, we can reconstruct the original nested structure. Note that we only do the reconstruction if there are no memory dependencies. This pass helps in case the new inner loop is vectorizable. Firstly, suppose there are any ϕ instructions other than the induction variable. In that case, we find the pattern discussed in the loop flattening pass to identify the ϕ instructions initially in the inner and outer loops. If there are no ϕ instructions from the outer loop, we switch the loop order to eliminate the ϕ instructions in the inner loop and make it vectorizable.

We create a new header and latch block, which mark the new outer loop. If we do not switch the order of the loops, we move the ϕ instructions known from the outer loop and delete the corresponding ϕ instructions created to keep track of the previous value. Next, we replace the outer count (i.e., the `udiv` for induction with the initial inner loop count in the flattened loop) with the induction from the new header and the inner count (i.e., `urem` value) with the inner induction. Finally, we update the branch instructions to correctly account for the outer and inner count.

If we want to switch the order of loops, for the inner ϕ instructions, we need to find if they are stored at a pointer plus `div` value (i.e., the original outer induction); if not, we allocate space equal to the outer loop count. We store the incoming value from the preheader in the corresponding pointer from zero to the outer loop count. For each ϕ instruction, add a store instruction in the new inner loop such that the incoming value from the loop latch is stored at the corresponding pointer plus induction and replace the ϕ instruction with a load from the corresponding pointer plus induction. This time, replace the `urem` value with the outer induction and the `div` value with the inner induction.

Appendix C. Loop Coalescing

We begin by identifying loops with distinct private conditions for their headers and latches, specifically targeting cases with private latches and corresponding maximum counts using `unroll_count`. This transformation focuses on nested loops with a depth of two, though it can be recursively applied to deeper nested loops. Let us denote the outer loop's components as preheader ph_o, header h_o, latch l_o, and block that exits the loop (exit block) e_o and the inner loop's components as preheader ph_i, header h_i, latch l_i and exit block e_i. Notably, a loop's header, latch, and exit blocks need not be distinct. We describe our transformation algorithm as follows:

1) **Create an empty loop:** We initialize an empty loop with header h_n and latch l_n. We set up an induction variable and loop condition such that the loop executes $2n + m$ times, where n is the maximum count for the outer loop and m for the inner loop. Additionally, we create two auxiliary basic blocks, b_1 and b_2.
2) **Maintain loop state:** In h_n, we use a ϕ instruction to maintain a state variable with `s=`ϕ`[1,`ph_o`][s',`l_n`]`, where `s'` is defined in l_n.
3) Below, we describe how the CFG is updated:
   - h_n: If s equals 1, h_n branches to h_o; otherwise, it goes to b_1.
   - ph_i: If ph_i unconditionally branches to h_i then it sets $s_{ph_i} = 2$; otherwise, let us denote the condition to branch to h_i as c_1, we set s_{ph_i} `= select` c_1 `2 3`. Finally, we update the terminator to an unconditional branch instruction to l_n.
   - b_1: If s equals 2, b_1 branches to h_i; otherwise, it proceeds to b_2.
   - b_2: If $s = 3$, b_2 branches to e; otherwise, to l_n.
   - l_i and e_i: If $l_i \neq e_i$ (then terminator must be unconditional branch to h_i), l_i sets $s_{l_i} = 2$ and branches to l_n unconditionally. Let us denote the exit condition in e_i with c_2, we set s_{e_i} `= select` c_2 `3 2`. If $e_i \neq l_i$, replace the successor so that it goes to l_n instead of the original exit; otherwise, we set the terminator as an unconditional branch to l_n.
   - l_o and e_o: Similarly, if $l_o \neq e_o$, l_o sets $s_{l_o} = 1$ and branches to l_n. If the exit condition for e_o is c_3, set s_{e_o} `= select` c_3 `-1 1`. If $e_o \neq l_o$, replace the successor such that it goes to l_n instead of the original exit; otherwise, we set the terminator as an unconditional branch to l_n.
4) Set `s' =` ϕ`[`s_{ph_i}`,`ph_i`][`s_{l_i}`,`l_i`][`s_{e_i}`,`e_i`][s,`b_2`][`s_{l_o}`,`l_o`][`s_{e_o}`,`e_o`]` in l_n. Note that l_i (l_o, resp.) is only used if $l_i \neq e_i$ ($l_o \neq e_o$, resp.).
5) **ϕ in h_o:** We move all the ϕ instructions in h_o to h_n and replace the incoming block from l_o to l_n.
6) **ϕ in h_i:** For each v `=` ϕ`[v1,`ph_i`][v2,`l_i`]` in h_i, we create v_l in l_n and v_i in h_n, such that v_i `=` ϕ`[`v_l`,`l_n`][v1,`ph_o`]` and v_l `=` ϕ`[v1,`b_2`][v1,`ph_i`][`v_l`,`l_o`][v2,`l_i`][v1,`e_o`][v1,`e_i`]` if $e_i \neq l_i$ and $e_o \neq l_o$. We replace all uses of v with v_i and erase v.
7) **Restore dominance relationship between instruction and users:** Let us say **b** is the set of blocks in the original loop that goes to l_n; we classify the basic blocks into parts such that each part corresponds to one block in **b** such that the blocks in a part are dominated by this predecessor block but do not dominate the next key block. Given these parts, we find the instructions used outside the part and add ϕ instructions to l_n to propagate the values.

Appendix D. End-to-End Integration

For cases where the MPC library uses its own classes corresponding to the cleartext data-types, e.g., `class::Integer` for `int`, we only need some additional functions in the interface in order to integrate the program with the MPC backend. These additional functions include the following:

- **Initialize:** Assuming a function-level integration, we first need initialization functions that take the input arguments and return the corresponding MPC object pointer. For arguments that are pointers to elementary data types we need to obtain additional metadata regarding the length of the arrays; this information cannot always be determined with certainty through program analysis.
- **Memory management:** For any `malloc` calls or `alloca` instructions, we add creator functions that allocate memory for the corresponding object and return the pointer. Similarly, for `free` calls we add deallocator functions that take object pointer and free the corresponding memory. These creator and destroyer functions are needed separately because the class definition is not known to the program before the linking stage thus, the size is unknown.
- **Memory access:** Since the size of objects is unknown, we need to create an interface for the `gep` (get element pointer) instruction that takes the pointer and an offset index, this function returns the corresponding pointer to the object at the input pointer plus offset. For `load` (resp. `store`) instructions, we can create an interface that takes the pointer (and object to store) and returns the object at the pointer. However, this can potentially lead to errors because compilers sometimes optimize the functions to follow `sret` format, implying that instead of returning large objects or classes, the function interface takes the first argument as a pointer to the return value and the

memory needs to allocated by the caller. Thus, to avoid such complications, we only deal with object pointers, and thus, our `store` interface looks like `store(Integer *ptr, Integer *val)` and we do not need a load interface.

- **Other differences:**
  - **Vector Instructions:** For vector instructions, all the interfaces remain mostly the same because they already handle operations at the pointer level; the only difference is that we must use the right object pointer instead of the pointers for the original cleartext data.
  - **Scalar Instructions:** For scalar instructions, we need to use pointers to objects (heap memory) instead of objects in the stack. For instance, for `mul` instruction, the interface is as follows:

    ```
    Integer *mul(Integer *a, Integer *b)
    ```

    While replacing the instructions with function calls this way, we cannot directly update the uses because the data types do not match. Thus, we use a map to store the original instruction versus the new instruction (object pointer). For each instruction, when creating the new function call for each operand, we use the corresponding object pointer from the aforementioned map.

D.1. Module Level Integration

At the module level, we need to update the program such that the object pointer corresponding to the return values can be used later, and the object pointers for the input arguments can be passed if they already exist. A naive solution to passing the corresponding object pointers is to change the function definitions; however, if the program itself is to be used as a library, this could lead to undesirable library interface changes. We observe that at any point, we only need to store the argument object pointers for at most one function that is being called, and once initialized, this extra memory that stores the corresponding pointers can be cleared. Note that this acts like an addition to the function stack, which stores the implicit argument object pointers and, after initialization, stores these pointers in its own stack, thus allowing us to clear this additional space.

For the output pointers, we note that all the new instructions deal with heap memory; thus, instead of freeing the corresponding object pointer, we store this pointer in a global pointer maintained by the MPC interface. On each call to the function, we load from this global pointer and use this in our map for plaintext to the MPC pointer.

Appendix E. Evaluation

E.1. Benchmark Suite

For all the benchmarks, we assume that the data is secret-shared between the parties. Below, we describe each benchmark in the benchmark suite:

1) **Biometric Matching:** We use a database with $N = 4096$ entries and each entry has $D = 4$ dimensions.
2) **Convex Hull:** We use a polygon of $N = 256$ vertices.
3) **Count10*:** Parties use a string a of $N = 4096$ and obtain the number of substrings that match the regex $10*$.
4) **Count10*2:** Parties use a string a of $N = 4096$ and obtain the number of substrings that match the regex $10 * 2$.
5) **DB Variance:** Parties compute the variance using a database with $N = 4096$.
6) **Histogram:** Parties use a dataset with $N = 4096$ entries to obtain a histogram with $D = 5$ bars.
7) **Inner Product:** Parties compute the inner product of two vectors of length $N = 4096$.
8) **k-Means Iteration:** Parties run one iteration of the k-Means clustering algorithm where the size of the input is $N = 256$ and the number of clusters is $k = 8$.
9) **Longest10*2:** Parties use a string a of $N = 4096$ and obtain the length of the largest substring that matches the regex $10 * 2$.
10) **Maximum Distance Between Symbols:** Parties compute the maximum distance between the symbol s (secret-shared) in a string of length $N = 4096$.
11) **MNIST ReLU:** Parties perform the MNIST ReLU computation over an array of $N = 4096$ elements.
12) **PSI:** Parties compute the PSI of two sets of length $N = 1024$.

E.2. Comparison With Prior Works

Table 6 and Table 7 show a detailed comparison of the time and memory usage by the compilers for each benchmark.