Modular Augmentation of LLVM for MPC

Radhika Garg · Xiao Wang · Northwestern University

How do we write code for MPC applications?

How do we write code for MPC applications?

EMP-toolkit, MOTION, ABY, …

- **Programmer directly calls MPC protocol functions**

  Full control — but requires deep cryptography expertise

- **No abstraction over private vs public data**

  Must manually track which variables are secret-shared

- **No compiler support whatsoever**

  No error messages, no optimization, no language features

- **Lowest-level interface to MPC**

  Appropriate for cryptographers building custom protocols

How do we write code for MPC applications?

Obliv-C, ObliVM, COMBINE, CBMC-GC, MP-SPDZ, Viaduct, Symphony, ...

- **Domain-specific language or added syntax**

  Programmer learns new annotations or a new language entirely

- **Compiler handles obliviousness automatically**

  But builds a custom toolchain — error messages and optimizations all from scratch

- **Cannot compose with legacy code**

  MPC code compiled separately; linking to existing C/C++ libraries is hard

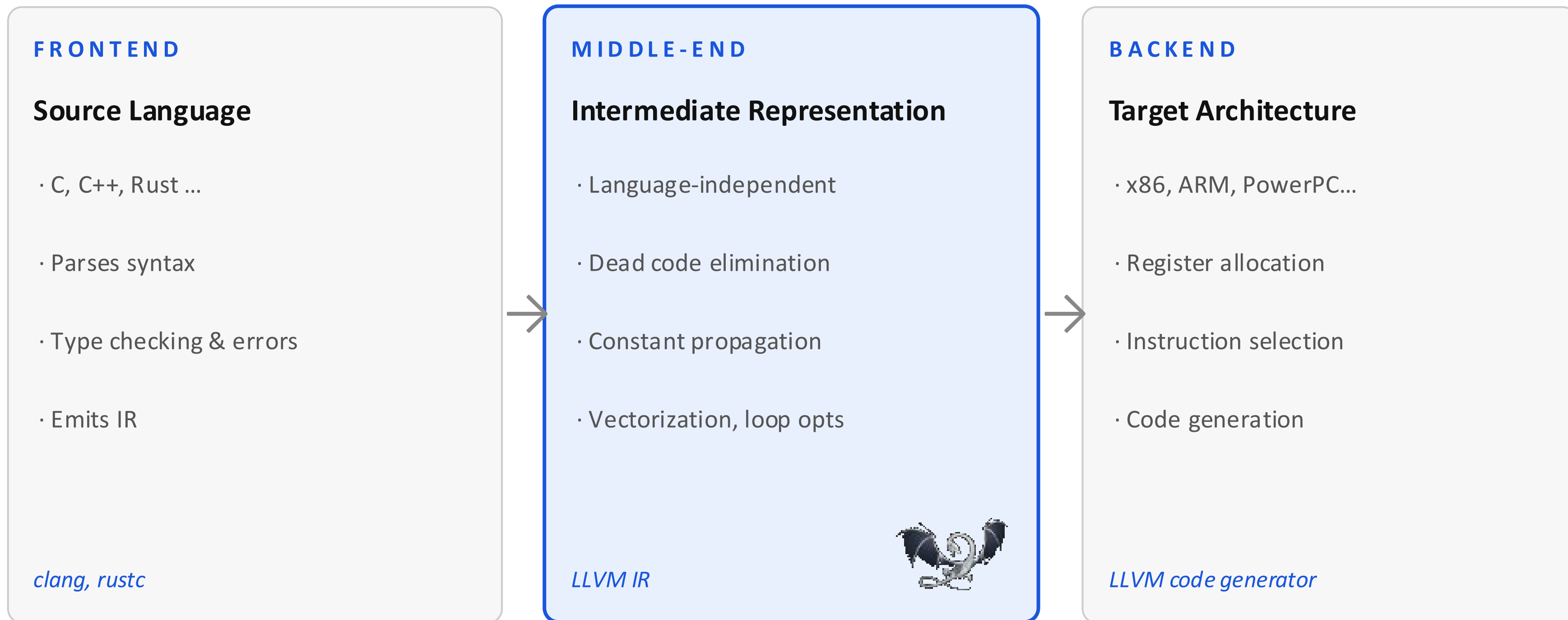
What about Hobbits?

Regular programmers who just want to build an application — not learn cryptography, not learn a new language.

- Usability: supports C, C++, Rust and any other LLVM-based language
- Avoids CPU emulation overhead
- Performance comparable/better than prior compilers and DSLs
- Ability to compose with legacy codebases
- Repurposes LLVM optimisations for MPC

How Does General-Purpose Computing Handle This?

A compiler separates concerns across three stages:



Key insight: the middle-end is shared across all languages and all targets.

Smaug: MPC at the IR Level

Smaug operates entirely as LLVM passes on LLVM IR — never builds a new compiler.

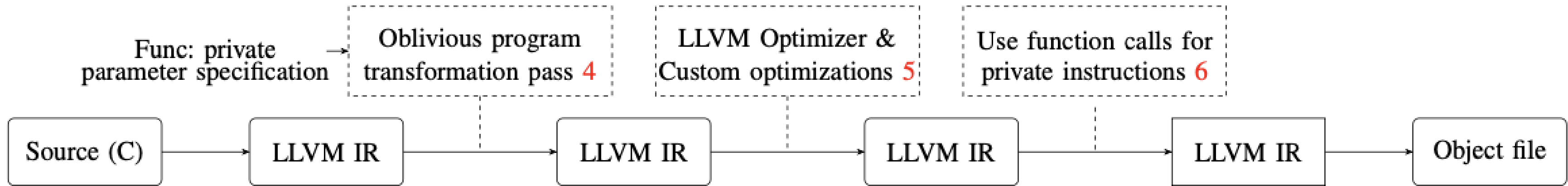


Figure 2: **Overview of Smaug.** The front end (Clang) converts the source (C) to LLVM IR. The IR can be optimized, for instance, by using -O1 with Clang. We transform the program (LLVM IR) given the specification for private function parameters for each function in a module into an oblivious program. 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 corresponding MPC function calls. Now, we use the LLVM code generator to compile the program to object code for the respective target.

Instead of a new language or a new compiler, Smaug adds LLVM passes that:

① Take a JSON spec identifying which function parameters are private

② Transform non-oblivious LLVM IR into its oblivious counterpart

③ Apply MPC-specific optimizations (vectorization for SIMD gates)

④ Replace private instructions with calls to the MPC library of your choice

Using Smaug: Step-by-Step

1 Write standard C++ (no new syntax)

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

2 JSON metadata: mark private args

```
{
  "histogram": {
    “inputs”: [<arg0_private>, <arg1_private>, …],
    “output”: <returning_private_value>
    “sizes”: [<ptr_arg_index>:<element_bits>, ..],
    “readAccess”:
[<ptr_arg_index>:<length_arg_index>, ..]
}
```

3 Compile

```
# 1. Source → LLVM IR
clang++ -I/usr/local/include -O0 -Xclang -disable-O0-optnone -S -emit-llvm -std=c++17 histogram.cpp -o histogram.ll

# 2. Run Smaug passes
opt --interleave-loops=false -load-pass-plugin=passes/build/libpasses.so \
    -passes=smaug-pipeline --metadata-path=histogram.ll.json --gc=true \
    histogram.ll -o histogram.ll -S

# 3. Link MPC runtime
clang++ -L/usr/local/lib -lssl -lcrypto -lemp-tool -maes -mssse3 -lgc histogram.ll -o histogram_mpc
```

The MPC Interface: Hooking IR Instructions to Protocols

A natural question: how does an LLVM instruction become a secure MPC operation?

LLVM IR INSTRUCTION

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

↓ *Smaug replaces with function call*

```
%c = call i32 @multI32(i32 %a, i32 %b)
```

↓ *Link with MPC library at compile time*

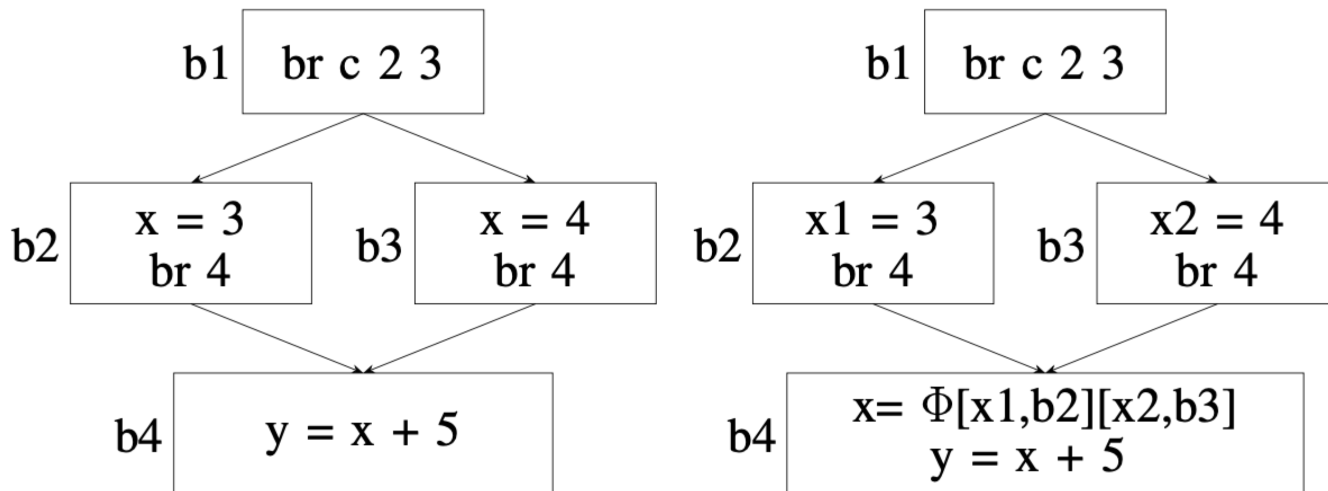
```
clang++ ... -lgc   histogram.ll -o
histogram_mpc
#            ^ swap for -lgmw for GMW backend
```

MPC INTERFACE (mpc.h — excerpt)

```
namespace MPC {

// Scalar operations on private values
int32_t multI32(int32_t a, int32_t b);
int32_t addI32 (int32_t a, int32_t b);
int32_t subI32 (int32_t a, int32_t b);
bool    icmpEqI32(int32_t a, int32_t b, int op);

// Vectorized (SIMD) operations
void multI32(int32_t *a, int32_t *b,
             int32_t *out, int n);
void addI32 (int32_t *a, int32_t *b,
             int32_t *out, int n);

// Utility
void setup(int party, int port);
void finish();
template<typename T> T reveal(T data, int p);

} // namespace MPC
```

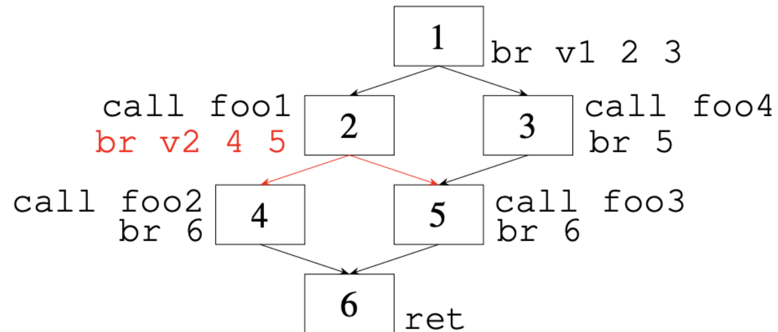
The interface is backend-agnostic — swap –lgc for -lgmw to change protocols.

SSA, PHI node background



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

The Challenge — Arbitrary CFGs with Mixed Branches



(...mpiler faces:

- Prior compilers restrict the source language
- They assume all branches are private, OR require a custom DSL
- Smaug takes unmodified LLVM IR — the CFG can have any structure after optimization
- A naive execute-every-block approach causes avoidable exponential blowup when branches are mixed public/private

2 **Example: mixed public/private branches**

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

v1 = **public** v2 = **private**

Key challenge:
Branch on v2 leaks a private value — but branch on v1 is safe to keep. Smaug must handle both in the same arbitrary CFG.

Non-Oblivious → Oblivious: Three-Step Transformation

Execute every reachable block — but make private branches invisible.

1

Compute execution conditions per block

- For each block get its condition variable.
- Split them in ‘private condition’ & ‘public’ ones — split them so we can handle each appropriately.

2

PHI nodes and store instruction

- Rewrite PHI nodes into a chain of select, (order matters for correctness)
- Update store instructions so no write commits unless the private condition says it should. No branch path is ever revealed.

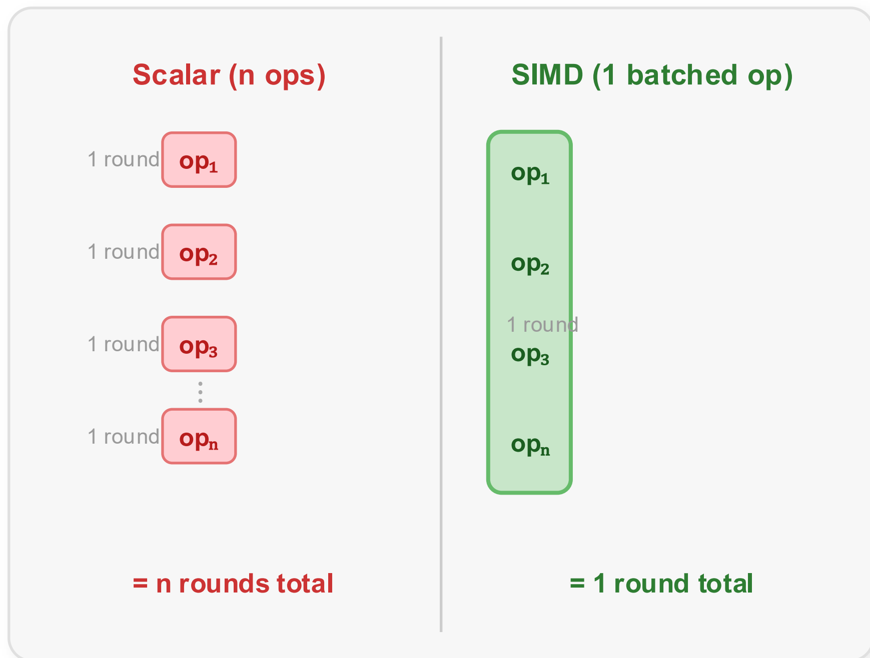
3

Restructure CFG on public conditions only

Remove private branches from the CFG entirely — only public structure remains. Avoids exponential blowup from naive linearization.

MPC specific optimization (batching)

MPC operations are expensive — communicating in rounds dominates cost.



1 The Core Insight

- Each MPC operation requires a round of network communication between parties.
- **n independent ops sequentially = n rounds.**
- **Same n ops batched with SIMD = 1 round.**
- Smaug's goal: automatically restructure your program so private operations happen in batches, not one at a time.

How Smaug Vectorizes: Steering LLVM

LLVM already vectorizes — Smaug just makes it MPC-aware.

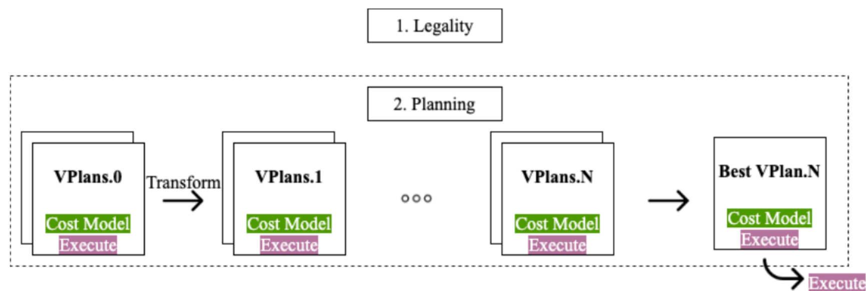


Figure 4: **Overview of LLVM Loop Vectorizer.** The LLVM

2 Two Nudges Smaug Adds

Updated Cost Model

LLVM treats private ops as cheap scalars by default. Smaug sets their cost to maximum when scalar, minimum when vectorized — so LLVM always chooses SIMD for private values.

Loop Splitting

If a loop fails LLVM's legality check (mixed vectorizable + non-vectorizable private ops), Smaug splits it. The vectorizable part passes legality and gets batched. Reductions are automatically parallelized into a single function call by LLVM.

Results: Compilation Overhead

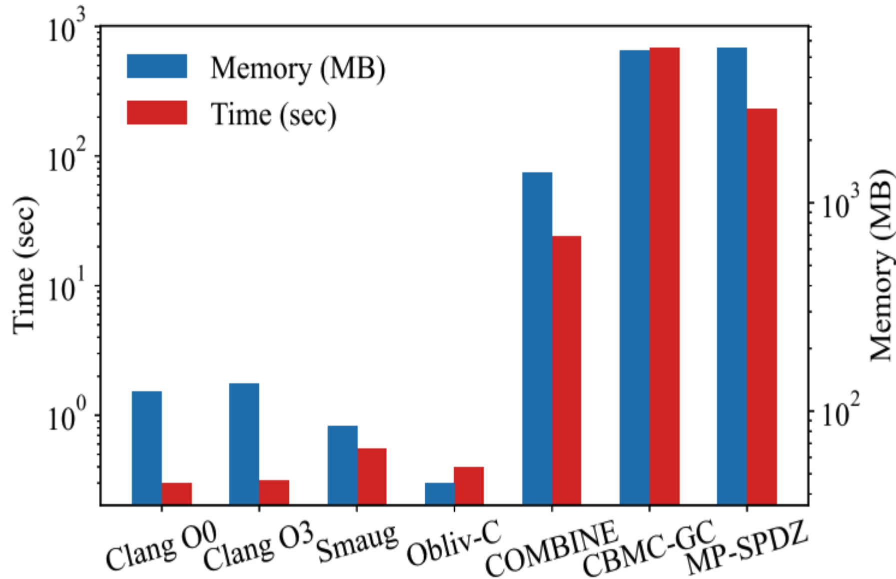


Figure 6: **Comparison of average time and memory usage.** Note that we are unable to compile some programs with CBMC-GC, and thus, they are omitted from their average.

Avg compilation time:

44× faster than COMBINE (avg)

1240× faster than CBMC-GC (avg)

420× faster than MP-SPDZ (avg)

≈ 1.4× slower than Obliv-C (no optimizations in Obliv-C)

Memory: 16× less than COMBINE · 63× less than CBMC-GC · 65× less than MP-SPDZ

Results: Runtime Performance

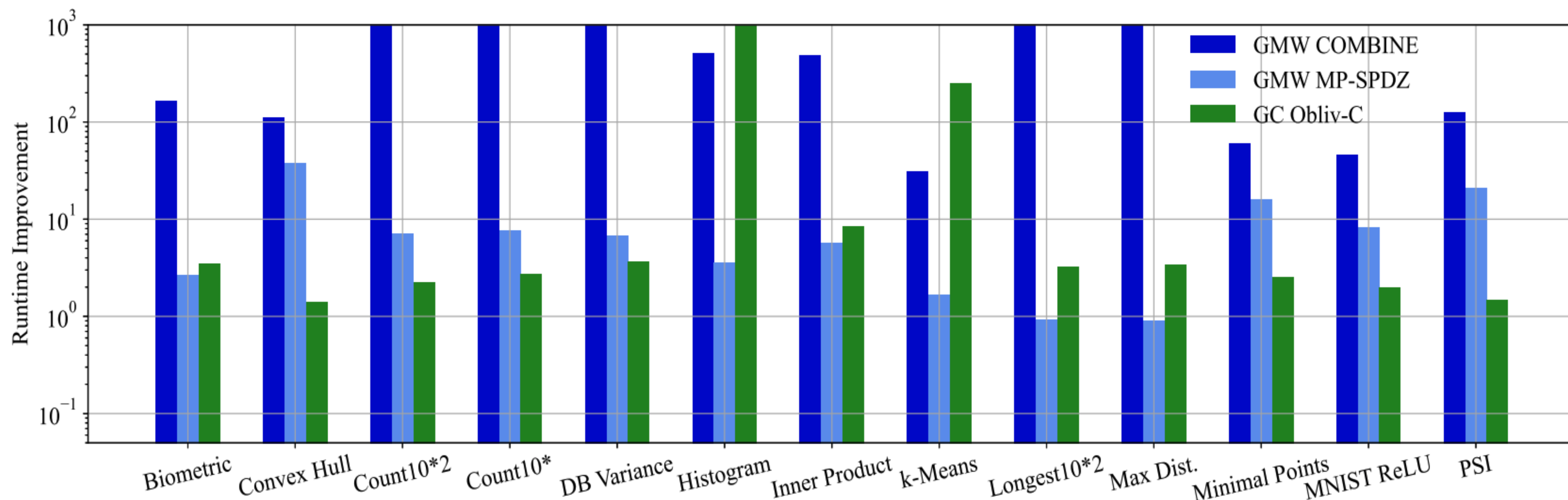


Figure 7: **Runtime improvement for programs compiled by COMBINE, MP-SPDZ, and Obliv-C vs Smaug.** DB-Variance, Longest10*2, and maximum distance compiled by COMBINE are killed before the process finishes. Histogram does not terminate when compiled with Obliv-C for the input size.

157× faster than COMBINE (biometric, GMW)

2× faster than MP-SPDZ (biometric, GMW)

1.5–8× faster than Obliv-C (GC backend)

Results: Circuit Size & Depth

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.

Key observations:

- ✅ GMW circuit size smaller than COMBINE and MP-SPDZ in most benchmarks
- ✅ GC circuit size closely matches Obliv-C (DSL baseline)
- ✅ Circuit depth: 570× lower than COMBINE on histogram — LLVM vectorization repurposed for SIMD
- ⚠️ CBMC-GC gets lower depth on some benchmarks but cannot compile many programs (memory OOM)

Real-World Programs — No New Language

K-Means Clustering

C — open-source, unmodified

Classifies secret-shared data between two parties stored in a MySQL database. Uses the standard C++ MySQL library — no MPC-specific code in the clustering library itself.

GC **23s**
GMW **28s**

Minesweeper

C++ — open-source, minimal changes

Mine locations private to the dealer; player's moves stay hidden from the dealer. Average response time per move on an 8×8 grid.

GC **0.19s**
GMW **0.21s**

Blackjack

C++ — open-source, minimal changes

Randomness shared between dealer and player — neither knows the deck order. Average response time after each move.

GC **7ms**
GMW **3ms**

Summary

No new language needed

Standard C/C++/Rust programs become MPC-enabled with a JSON spec. Users keep their existing tools and skills.

LLVM gives you everything for free

Error messages, optimizations, multi-language support, active maintenance — none of this has to be built.

Compilation is orders of magnitude faster

44× vs COMBINE, 1240× vs CBMC-GC, 420× vs MP-SPDZ. Scales independently of input size.

Runtime matches or beats DSL tools

157× vs COMBINE, 1.5–8× vs Obliv-C. Circuit size matches Obliv-C and often beats COMBINE and MP-SPDZ.

Real programs just work

Minesweeper, Blackjack, K-Means + MySQL — compiled with minimal changes, no MPC expertise required from the user.

Thank You

Paper: https://eprint.iacr.org/2025/004.pdf

Code: https://github.com/radhika1601/smaug

Questions?