# Multi-Precision Arithmetic for Cryptography in C++

*at Run-Time and at Compile-Time*

Niek J. Bouman, PhD

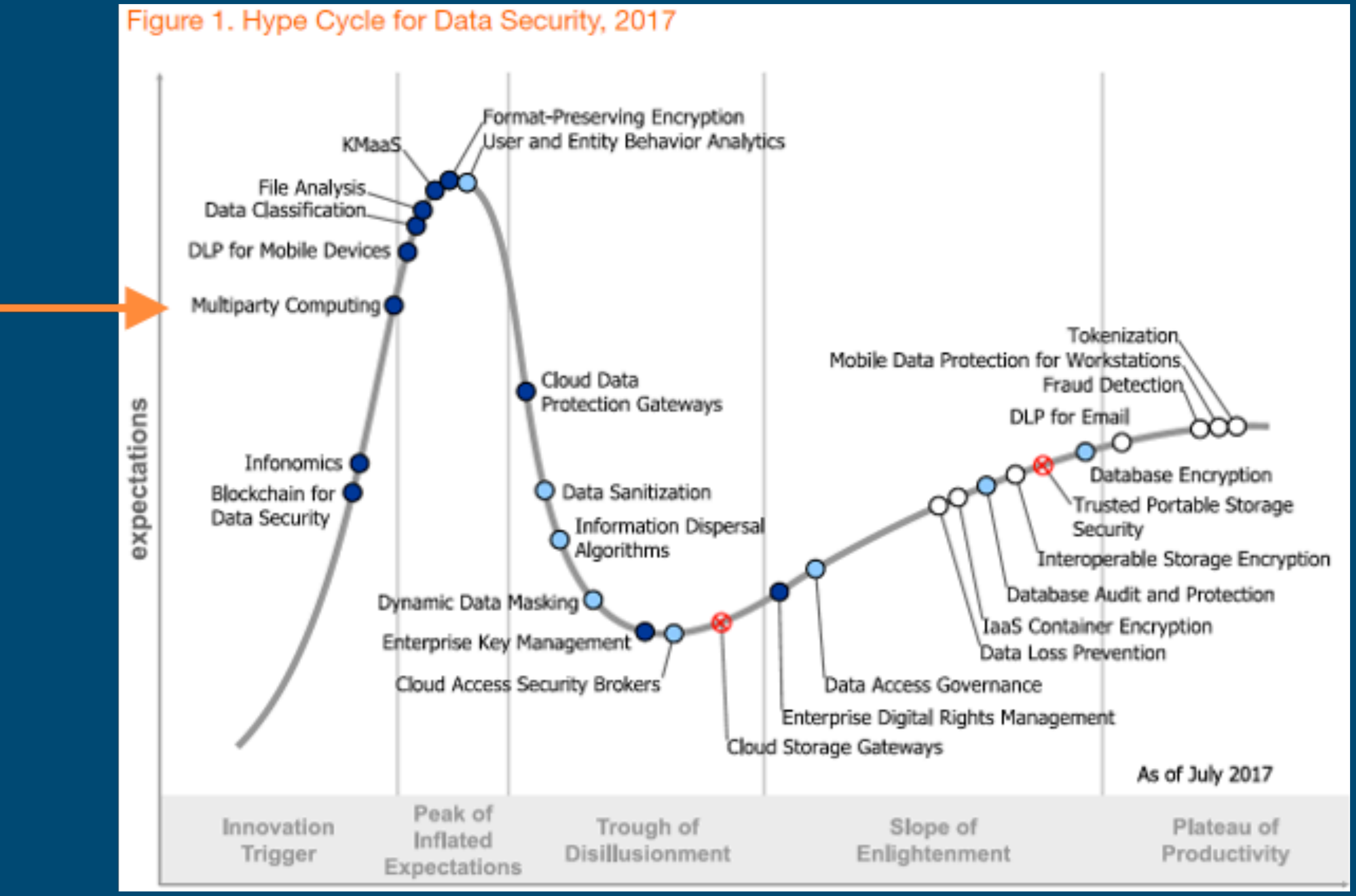
niekbouman@gmail.com

cppcon the c++ conference
SEPTEMBER 23-29 2018
Bellevue, Washington, USA

Scalable Oblivious Data Analytics

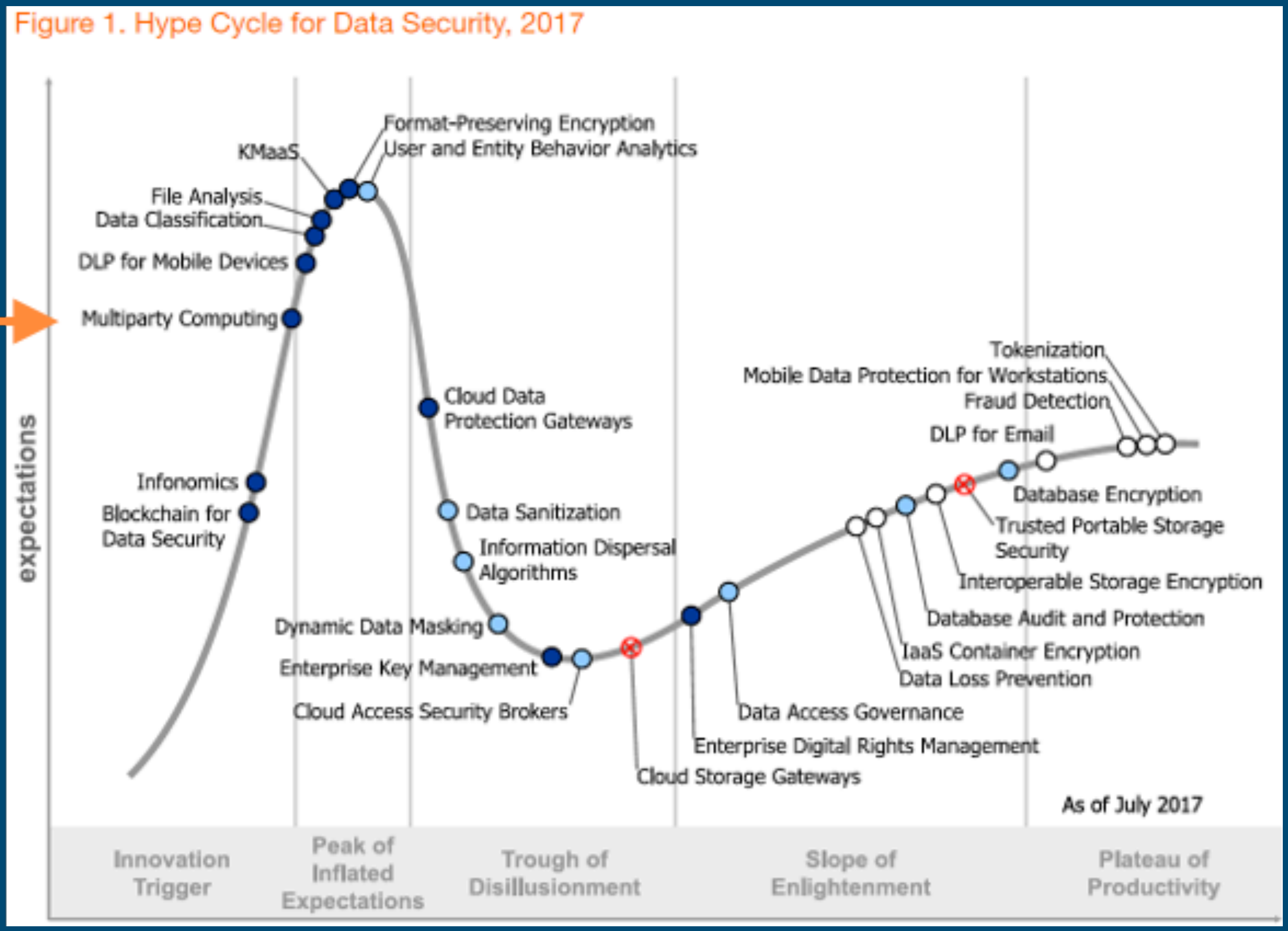
Technische Universiteit Eindhoven
University of Technology

# Secure Multiparty Computation (MPC)




**...in Gartner's Hype Cycle (2017)**

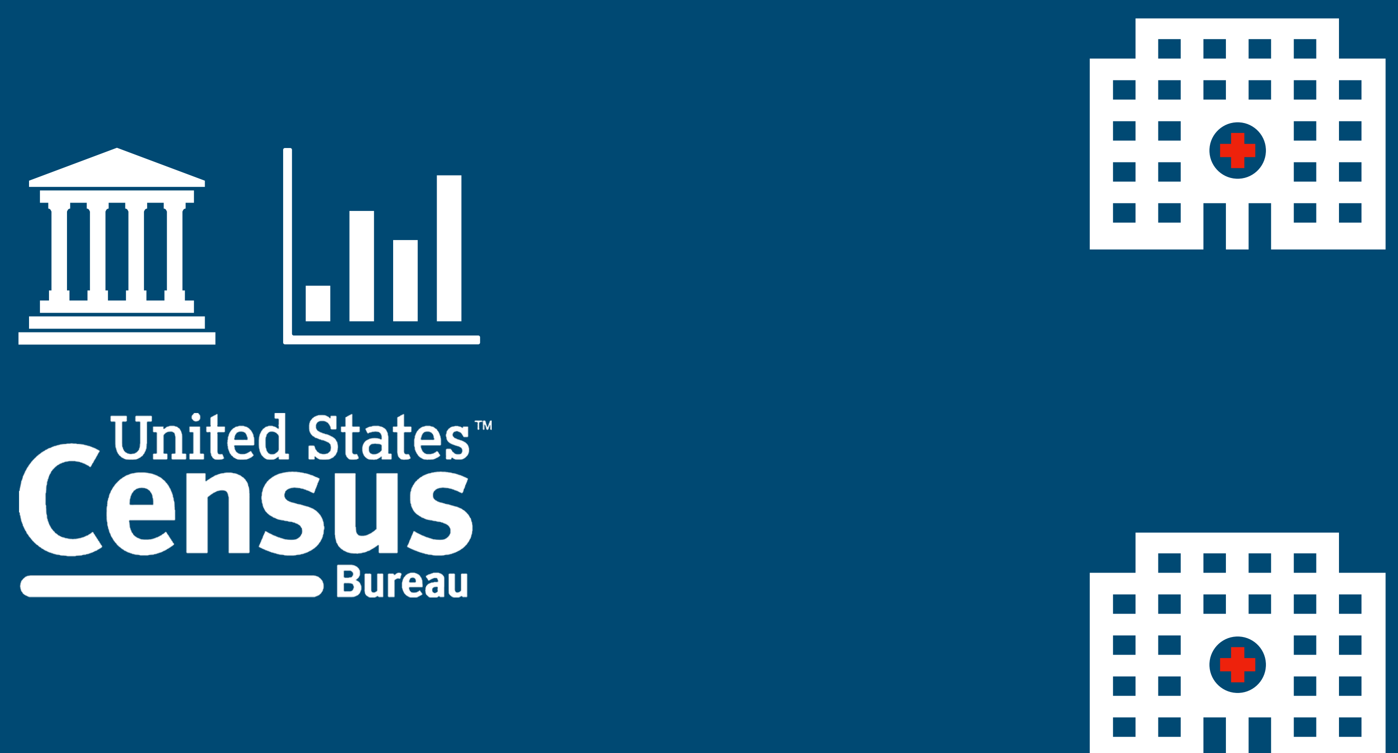
# Secure Multiparty Computation (MPC)




**...in Gartner's Hype Cycle (2017)**

- $N$ parties jointly compute a function $f(x_1, \ldots, x_N)$ such that no party learns anything beyond the output
- In essence, the $N$ parties *jointly emulate* a "virtual trusted party" that computes $f$ for the parties

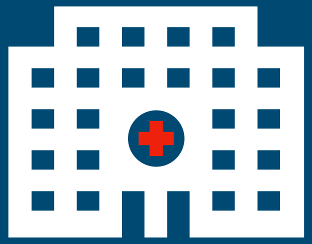
# Example: Privacy-Preserving Data Analysis

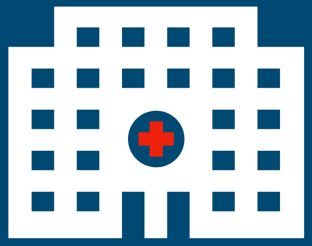


# Example: Privacy-Preserving Data Analysis

- Instead of sending all patient data to the Census Bureau, the hospitals and the Census Bureau run an MPC protocol
- Census Bureau only learns about the desired statistic, without seeing the underlying data

United States™ Census Bureau





# Big-Integer Arithmetic for Cryptographic Applications

**Target Size ≈ 200 Bits**

# Big-Integer Arithmetic for Cryptographic Applications

Secure Multiparty Computation

Target Size ≈ 200 Bits

Elliptic Curve Cryptography

# Big-Integer Arithmetic

Big-number arithmetic

- numbers do not fit in a single (64-bit) register

# Big-Integer Arithmetic

Big-number arithmetic

- numbers do not fit in a single (64-bit) register

Required operations

- addition: $a + b$
- multiplication: $a \times b$

# Big-Integer Arithmetic

## Big-number arithmetic

- numbers do not fit in a single (64-bit) register

## Required operations

- addition: $a + b$
- multiplication: $a \times b$
- addition mod $q$: $(a + b) \,\%\, q$
- multiplication mod $q$: $(a \times b) \,\%\, q$
- in our application, $q$ is in the order of 200 bits (and $q$ is prime)

# Big-Integer Arithmetic

Big-number arithmetic

- numbers do not fit in a single (64-bit) register

Required operations

- addition: $a + b$
- multiplication: $a \times b$
- addition mod $q$: $(a + b)\ \%\ q$
- multiplication mod $q$: $(a \times b)\ \%\ q$
- in our application, $q$ is in the order of 200 bits (and $q$ is prime)
- (and other operations: like comparison, ...)

# Yet Another Big-Int Library?

# Yet Another Big-Int Library?

- $q$ has fixed size & all numbers smaller than $q$

  $\Rightarrow$ fixed-size storage: `std::array`

# Yet Another Big-Int Library?

- $q$ has fixed size & all numbers smaller than $q$

  $\Rightarrow$ fixed-size storage: std::array

- header only => compiler has access to the implementation of functions

- $q$ known at compile-time

  - leverage compile-time optimizations

# Yet Another Big-Int Library?

- $q$ has fixed size & all numbers smaller than $q$

  $\Rightarrow$ fixed-size storage: std::array

- header only => compiler has access to the implementation of functions
- $q$ known at compile-time
  - leverage compile-time optimizations
  - in order to do this, we need to perform
    big-integer computations during compile-time...
    let's use constexpr

Example: The compiler can rewrite a modulo operation with a fixed modulus into a multiplication followed by a bitshift.

We want the same optimisation for big-ints. This requires the ability to perform computation with big-ints at compile-time.

```
unsigned long foo(unsigned long a)
{
    return a % 78623419;
}
```

```
foo(unsigned long):
        movq    %rdi, %rcx
        movabsq $-2701562103368370257, %rdx
        movq    %rcx, %rax
        mulq    %rdx
        shrq    $26, %rdx
        imulq   $78623419, %rdx, %rax   # i
        subq    %rax, %rcx
        movq    %rcx, %rax
        retq
```

- header only => compiler has access
- $q$ known at compile-time
  - leverage compile-time optimizations
  - in order to do this, we need to perform big-integer computations during compile-time... let's use constexpr

# Yet Another Big-Int Library?

- $q$ has fixed size & all numbers smaller than $q$

  $\Rightarrow$ fixed-size storage: std::array

- header only => compiler has access to the implementation of functions

- $q$ known at compile-time

  - leverage compile-time optimizations

  - in order to do this, we need to perform big-integer computations during compile-time... let's use constexpr

# Yet Another Big-Int Library?

- $q$ has fixed size & all numbers smaller than $q$

  $\Rightarrow$ fixed-size storage: std::array

- header only => compiler has access to the implementation of functions

- $q$ known at compile-time

  - leverage compile-time optimizations

  - in order to do this, we need to perform big-integer computations during compile-time... let's use constexpr

- ...should have a "natural" Modern-C++ API, and be fast, bug-free, and constant-time

# “Natural” C++ API

# “Natural” C++ API

- Compile-time parsing of immediate values

```
auto x = 18446744073709551616_Z; //user-defined literal
```

# “Natural” C++ API

- Compile-time parsing of immediate values

```
auto x = 18446744073709551616_Z; //user-defined literal
// x has type std::integer_sequence<uint64_t, 6, 10>
```

# “Natural” C++ API

- Compile-time parsing of immediate values

```
auto x = 18446744073709551616_Z; //user-defined literal
// x has type std::integer_sequence<uint64_t, 6, 10>
```

- For arithmetic mod $q$, modulus gets “baked into” the numeric type (no overhead at runtime for storing the modulus)

```
using GF = decltype(Zq(885443715538058477627_Z)); // 70-bit prime
//GF has type ZqElement<uint64_t, 59, 48>
```

# “Natural” C++ API

- Compile-time parsing of immediate values

```
auto x = 18446744073709551616_Z; //user-defined literal
// x has type std::integer_sequence<uint64_t, 6, 10>
```

- For arithmetic mod $q$, modulus gets “baked into” the numeric type (no overhead at runtime for storing the modulus)

```
using GF = decltype(Zq(885443715538058477627_Z)); // 70-bit prime
//GF has type ZqElement<uint64_t, 59, 48>

GF x { 885443715538058477626_Z }; // q-1
GF y { 2_Z };
```

# "Natural" C++ API

- Compile-time parsing of immediate values

```
auto x = 18446744073709551616_Z; //user-defined literal
// x has type std::integer_sequence<uint64_t, 6, 10>
```

- For arithmetic mod $q$, modulus gets "baked into" the numeric type (no overhead at runtime for storing the modulus)

```
using GF = decltype(Zq(885443715538058477627_Z)); // 70-bit prime
//GF has type ZqElement<uint64_t, 59, 48>

GF x { 885443715538058477626_Z }; // q-1
GF y { 2_Z };

auto sum_mod_q = x + y; // sum_mod_q == 1_Z
// '+' is overloaded, such that it performs modulo reduction
```

# A Recurring Pattern with `std::integer_sequence`

1. Compute something, return result as `std::integer_sequence`

$$[\ \ast,\ \ast,\ \ast,\ \ldots,\ 0,\ 0,\ \ldots]$$

(unknown number of trailing zeros)

2. Count the number of trailing zeros and cut them off

- Easy, right?
- Implementation becomes rather messy... (next slide)

```cpp
template <typename T, T... Is>
constexpr auto tight_length(std::integer_sequence<T, Is...>){
  size_t L = sizeof...(Is);
  std::array<T, sizeof...(Is)> num {Is...};
  while (L > 0 && num[L - 1] == 0) --L;
  return L;
}

template <typename T, T... Limbs, size_t... Is>
constexpr auto take_first(std::integer_sequence<T, Limbs...>, std::index_sequence<Is...>) {
  constexpr std::array<T, sizeof...(Limbs)> num = {Limbs...};
  return std::integer_sequence<T, num[Is]...> {};
}

template <size_t N, typename T = uint64_t, size_t... Is>
constexpr auto actual_computation(std::index_sequence<Is...>) {
  constexpr std::array<T, N> working_storage {};
  ...
  return std::integer_sequence<T, working_storage[Is]...>{};
}

template <size_t N, typename T = uint64_t>
constexpr auto some_computation() {
  auto m = actual_computation<N>(std::make_index_sequence<N>{});
  return take_first(m, std::make_index_sequence<tight_length(m)>{});
}
```

# Parameter Passing: GMP's `mpn_add_n`

# Parameter Passing: GMP's mpn_add_n

```
mp_limb_t mpn_add_n (mp_ptr rp, mp_srcptr up, mp_srcptr vp,
                     mp_size_t n)
{
  mp_limb_t ul, vl, sl, rl, cy, cy1, cy2;
  cy = 0;
  do {
      // loop body
  } while (--n != 0);
  return cy;
}
```

# Parameter Passing: GMP's `mpn_add_n`

```
mp_limb_t mpn_add_n (mp_ptr rp, mp_srcptr up, mp_srcptr vp,
                     mp_size_t n)
{
  mp_limb_t ul, vl, sl, rl, cy, cy1, cy2;
  cy = 0;
  do {
      // loop body
  } while (--n != 0);
  return cy;
}
```

# Parameter Passing: GMP's `mpn_add_n`

```
mp_limb_t mpn_add_n (mp_ptr rp, mp_srcptr up, mp_srcptr vp,
                     mp_size_t n)
{
  mp_limb_t ul, vl, sl, rl, cy, cy1, cy2;
  cy = 0;
  do {
      // loop body
  } while (--n != 0);
  return cy;
}
```

- compiler cannot see link between $n$ and array lengths

# Parameter Passing: GMP's mpn_add_n

```
mp_limb_t mpn_add_n (mp_ptr rp, mp_srcptr up, mp_srcptr vp,
                     mp_size_t n)
{
  mp_limb_t ul, vl, sl, rl, cy, cy1, cy2;
  cy = 0;
  do {
      // loop body
  } while (--n != 0);
  return cy;
}
```

- compiler cannot see link between $n$ and array lengths

# Parameter Passing: GMP's `mpn_add_n`

```
mp_limb_t mpn_add_n (mp_ptr rp, mp_srcptr up, mp_srcptr vp,
                     mp_size_t n)
{
  mp_limb_t ul, vl, sl, rl, cy, cy1, cy2;
  cy = 0;
  do {
      // loop body
  } while (--n != 0);
  return cy;
}
```

- compiler cannot see link between $n$ and array lengths
- $n$ is not a compile-time value, no loop unrolling
- possibility of aliasing between inputs and output might block compiler optimizations

# Parameter Passing: our Library

# Parameter Passing: our Library

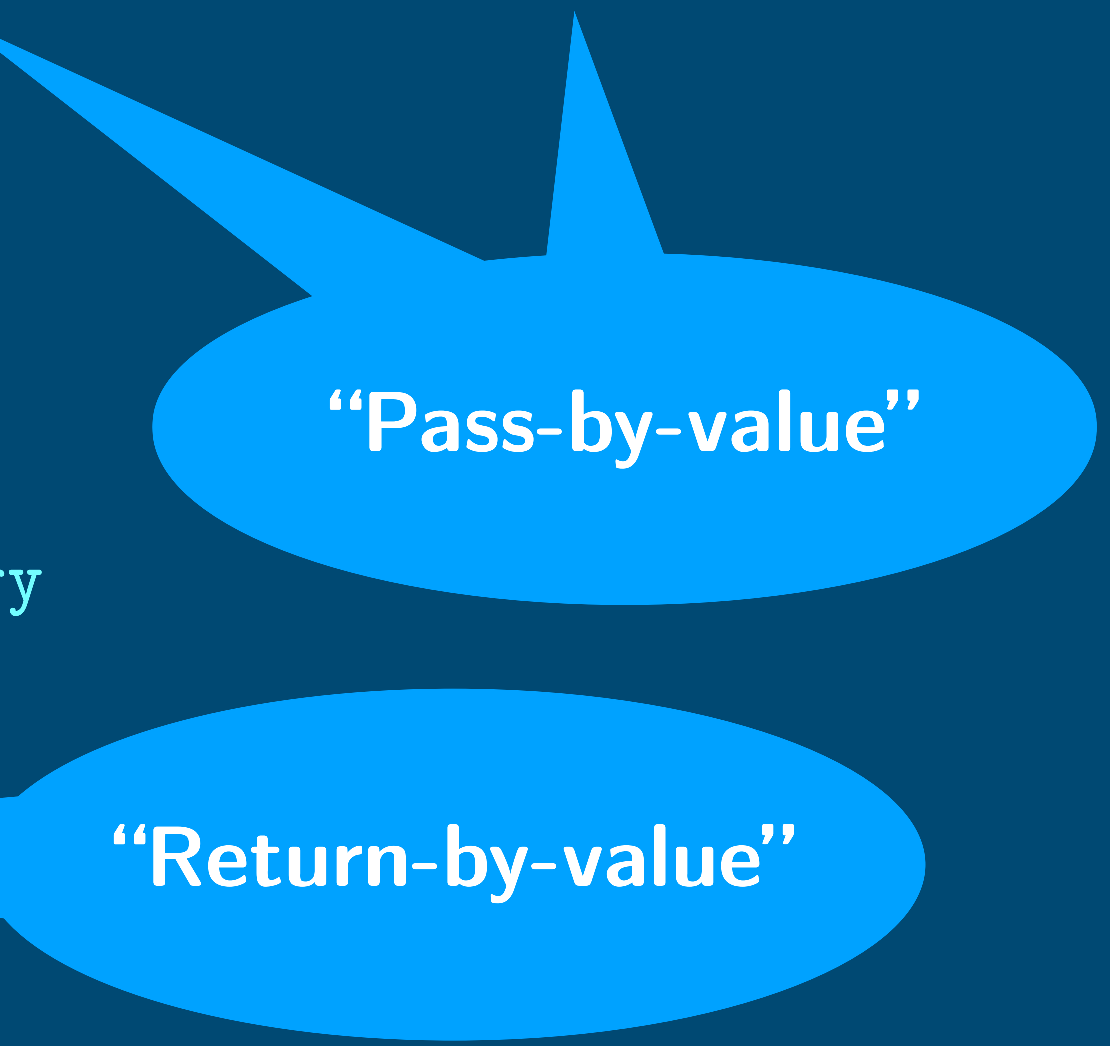
```
template <typename T, size_t N>
constexpr auto add(std::array<T, N> a, std::array<T, N> b)
{
  std::array<T, N+1> result {};

  auto carry = 0;
  for(auto i = 0; i < N; ++i) {
     // addition body
     // updates result[i] and carry
  }
  result[N] = carry;
  return result;
}
```

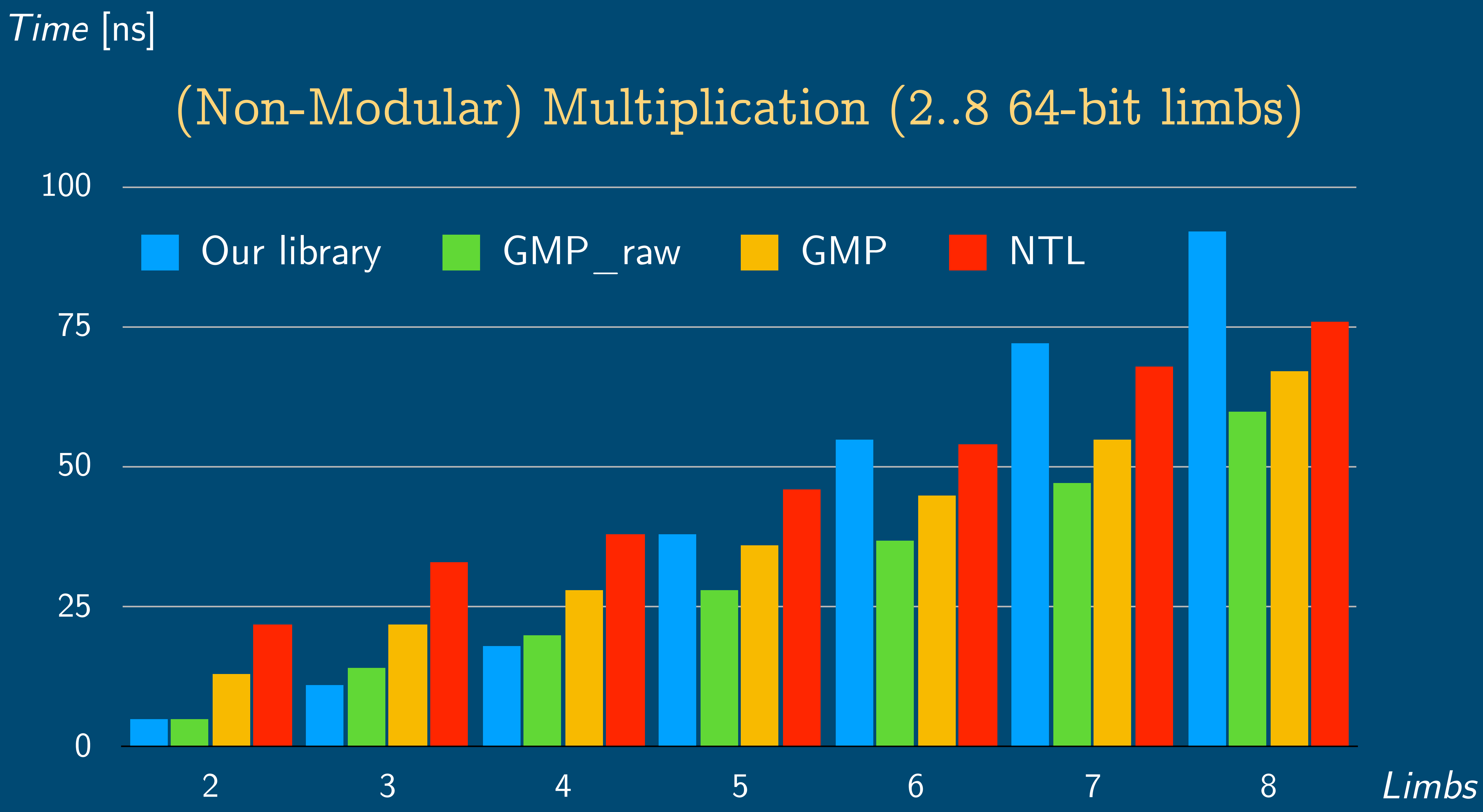
# Parameter Passing: our Library

```
template <typename T, size_t N>
constexpr auto add(std::array<T, N> a, std::array<T, N> b)
{
  std::array<T, N+1> result {};

  auto carry = 0;
  for(auto i = 0; i < N; ++i) {
     // addition body
     // updates result[i] and carry
  }
  result[N] = carry;
  return result;
}
```

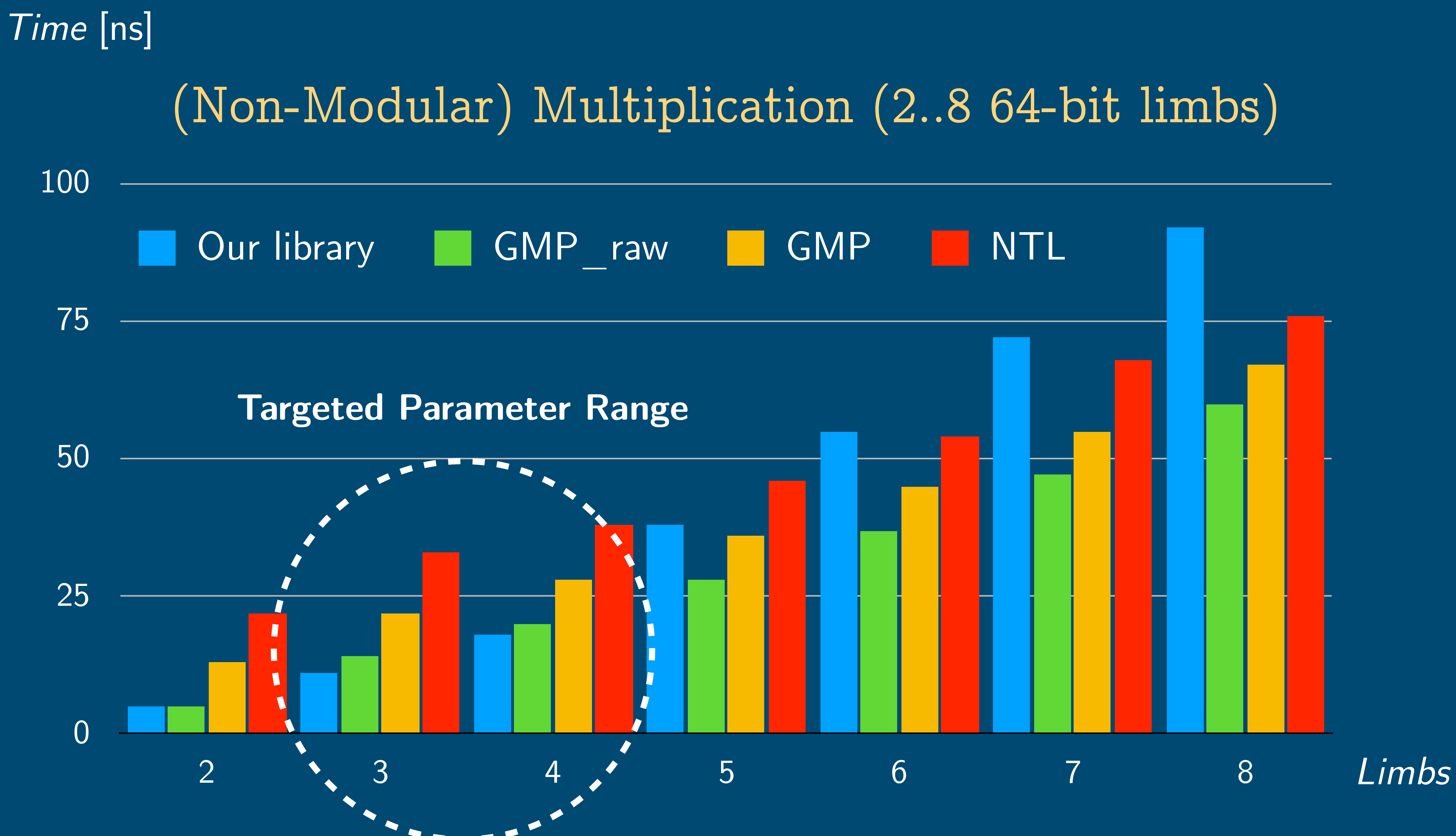


# Preliminary Benchmarks *(Google Benchmark lib)*




"GMP" means **mpn_mul**
"GMP-raw" means **__gmpn_mul_basecase** (not part of GMP's public API)

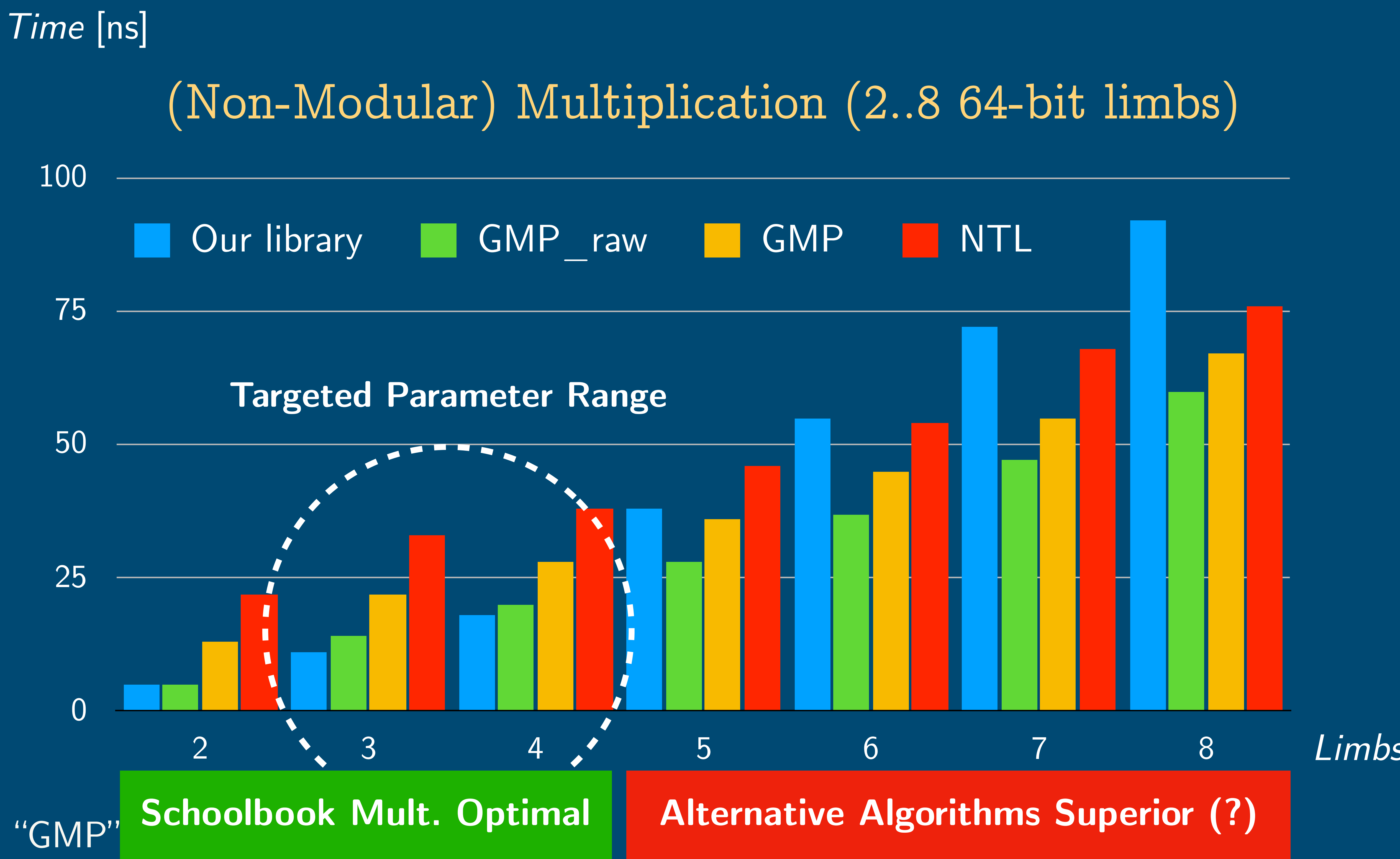
# Preliminary Benchmarks *(Google Benchmark lib)*




"GMP" means **mpn_mul**

"GMP-raw" means **__gmpn_mul_basecase** (not part of GMP's public API)

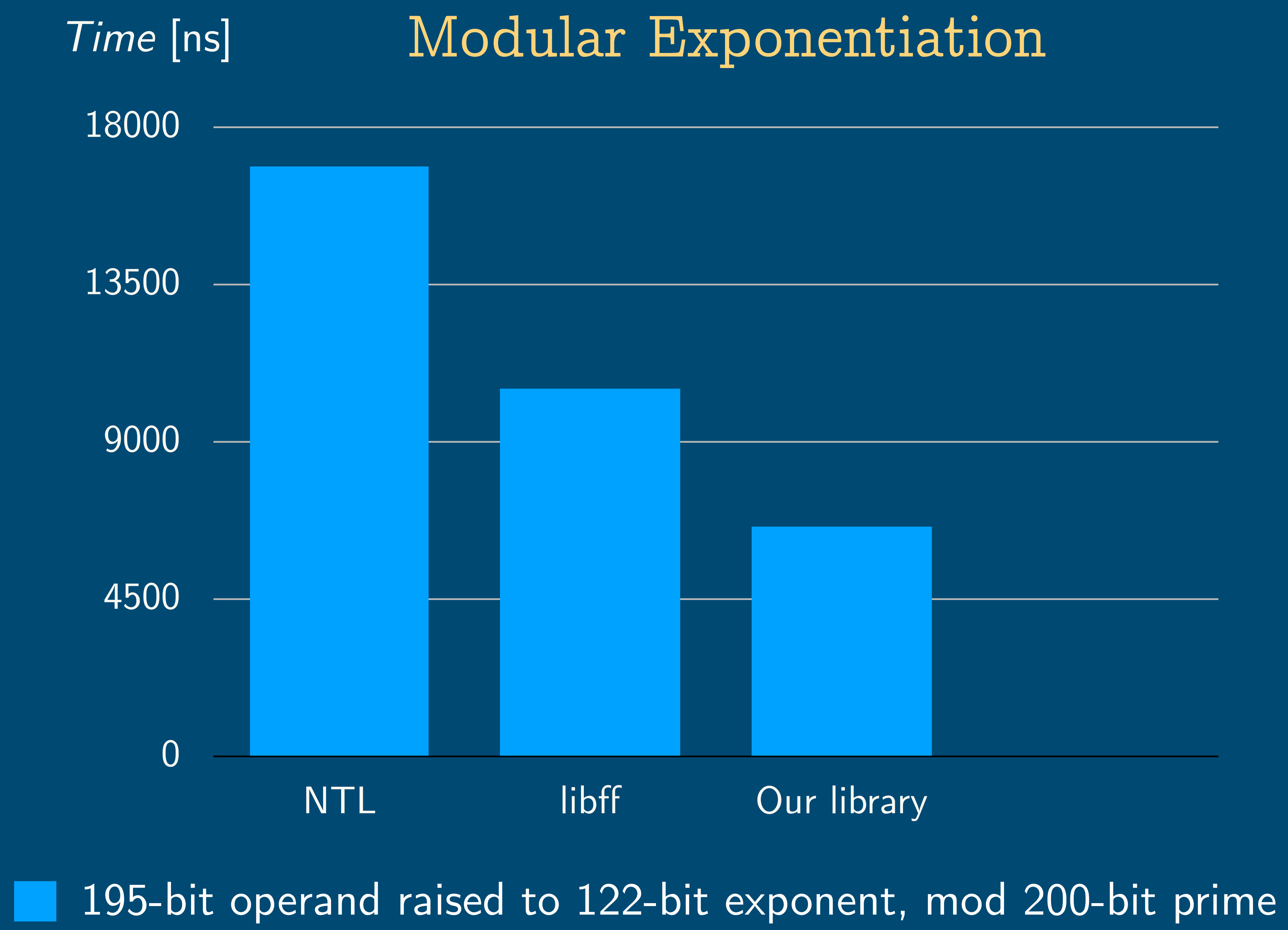
# Preliminary Benchmarks *(Google Benchmark lib)*




“GMP”

“GMP-raw” means **__gmpn_mul_basecase** (not part of GMP’s public API)

# Preliminary Benchmarks *(Google Benchmark lib)*



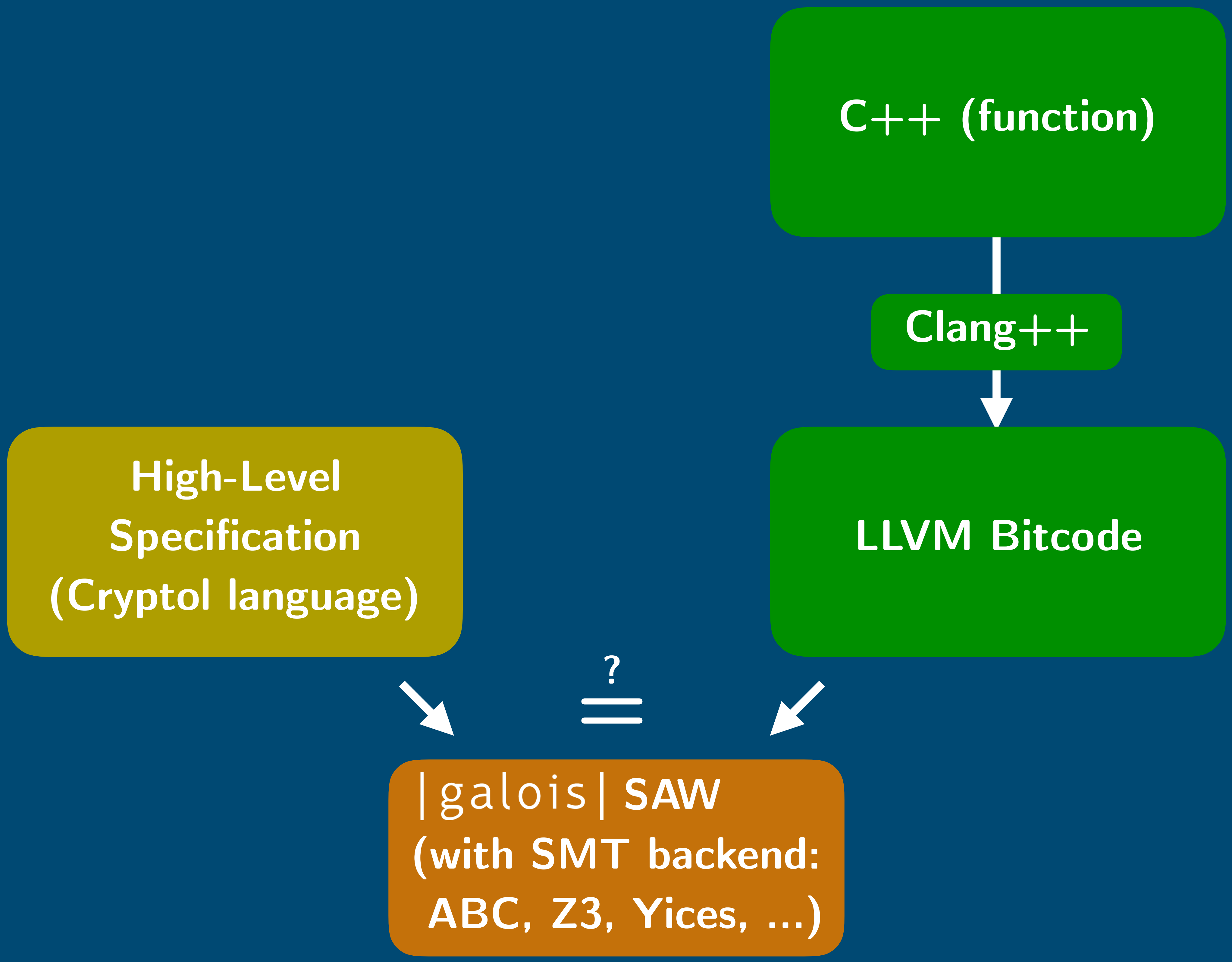
fast, bug-free, and constant-time

✓

# Bug-Free?

- Using existing tools, it is sometimes feasible (depending on the particular function) to obtain a formal correctness proof
- Prove equivalence between high-level specification (in a special functional language) and LLVM bitcode of our C++ implementation
- Gives the "best possible unit test"

# Correctness via Equivalence Proofs



# Example: Addition Spec in Cryptol

```
add_ref : {n, w} (fin n, fin w, n >= 1, w >= 1) =>
          [n][w] -> [n][w] -> [n+1][w]
add_ref a b = num_to_bigint`{n+1,w}
              (
    (bigint_to_num`{out=(n+1)*w} a) + (bigint_to_num`{out=(n+1)*w} b)
              )
```

# Example: Addition Spec in Cryptol

```
add_ref : {n, w} (fin n, fin w, n >= 1, w >= 1) =>
          [n][w] -> [n][w] -> [n+1][w]
add_ref a b = num_to_bigint`{n+1,w}
              (
    (bigint_to_num`{out=(n+1)*w} a) + (bigint_to_num`{out=(n+1)*w} b)
              )

num_to_bigint : {n,w} (fin n, fin w, n>=1, w>=1)
             => [n*w] -> [n][w]
num_to_bigint num = reverse (split`{n} num)

bigint_to_num : {n,w,out} (fin n, fin w, fin out, n>=1, w>=1, out >= n*w)
             => [n][w] -> [out]
bigint_to_num limbs = zero # (join (reverse limbs))
```

# Example: Equivalence Proof Setup in SAW

```
import "add.cry";
let mangled_name = "_ZN3cbn3addIyLm4EEEDaNS_7big_intIXT0_ET_NSt3__19
                    enable_ifIXsr3std11is_integralIS3_EE5valueEvE4typeEEES8_";

let my_alloc n ty ty2 = do { /* function body omitted*/ };

let add_spec n_ w_ = do {
    (xs, xp) <- my_alloc "xs" (llvm_struct "struct.cbn::big_int.0")
(llvm_array n_ (llvm_int w_));
    (ys, yp) <- alloc_and_bind "ys" (llvm_struct "struct.cbn::big_int.0")
(llvm_array n_ (llvm_int w_));
    (rs, rp) <- alloc_and_bind "rs" (llvm_struct "struct.cbn::big_int")
(llvm_array (eval_int {{ ((`n_):[8]) + 1 }}) (llvm_int w_));
    crucible_execute_func [rp, xp, yp];
    crucible_points_to rp (crucible_term {{ add_ref`{n=n_,w=w_} xs ys }});
};

m <- llvm_load_module "add.bc";
add_ov <- crucible_llvm_verify m mangled_name [] true (add_spec 4 64) z3;
```

# Example: Equivalence Proof Setup in SAW

**Load Cryptol spec**

```
import "add.cry";
let mangled_name = "_ZN3cbn3add1yLm4EEEDaNS_7big_intIXT0_ET_NSt3__19
                    enable_ifIXsr3std11is_integralIS3_EE5valueEvE4typeEEES8_";

let my_alloc n ty ty2 = do { /* function body omitted*/ };

let add_spec n_ w_ = do {
```

**Describe data types**

```
    (xs, xp) <- my_alloc "xs" (llvm_struct "struct.cbn::big_int.0")
(llvm_array n_ (llvm_int w_));
    (ys, yp) <- alloc_and_bind "ys" (llvm_struct "struct.cbn::big_int.0")
(llvm_array n_ (llvm_int w_));
    (rs, rp) <- alloc_and_bind "rs" (llvm_struct "struct.cbn::big_int")
(llvm_array (eval_int {{ ((`n_):[8]) + 1 }}) (llvm_int w_));
```

**Describe how to call our function**

```
    crucible_execute_func [rp, xp, yp];
```

**Require that output coincides with Cryptol spec**

```
    crucible_points_to rp (crucible_term {{ add_ref`{n=n_,w=w_} xs ys }});
};
```

**Load LLVM bitcode**

```
m <- llvm_load_module "add.bc";
```

**Set various parameters and run verifier**

```
add_ov <- crucible_llvm_verify m mangled_name [] true (add_spec 4 64) z3;
```

# Correctness Via Equivalence Proofs

...with varying success

- proof for `add` function found in matter of seconds
- prover does not terminate for `mul` function (on realistic input sizes)
  => requires another approach

fast, bug-free, and constant-time

✓ ✓?

# Constant-Time?

- A function enjoys the constant-time property if it does not "leak" any information about values designated as secrets (e.g., the secret key in an encryption function) via timing side-channels
- Attacker cannot infer any secret information from measuring execution time of such function

# Constant-Time: don’ts...

- Don’t branch on a secret value

```
if (secret_bit) {
    //...
}
```

- Don’t index memory with a secret value

```
// ‘arr’ is a pointer to an array
auto x = arr[secret_int];
```

# Automatic Constant-Time Verification

- `ct-verif` — tool for constant-time verification of C programs
  sound and complete
  (Almeida, Barbosa, Barthe, Dupressoir, Emmi, 2016)

- Verification on the level of optimized LLVM bitcode
  (step from LLVM IR to target-CPU-specific code currently remains unverified)

- Uses Boogie (Microsoft Research) as a backend

# Automatic Constant-Time Verification

- `ct-verif` — tool for constant-time verification of C programs
  sound and complete
  (Almeida, Barbosa, Barthe, Dupressoir, Emmi, 2016)
- Verification on the level of optimized LLVM bitcode
  (step from LLVM IR to target-CPU-specific code currently remains unverified)
- Uses Boogie (Microsoft Research) as a backend
- Via some minor modifications, now also works for C++!

**https://github.com/niekbouman/verifying-constant-time/tree/cplusplus**

# Running ct-verif

```
niek@macwin023 $ docker exec --env 'BOOGIE=monodocker exec --env 'BOOGIE=mono //
boogie/Binaries/Boogie.exe' --workdir /test -it 7f41630821d2 /verifying-constant-
time/bin/ct-verif.rb --clang-options '-x c++ -std=c++14 -O3 -I ctbignum/include
-Wno-c++1z-extensions' -e _Z15generic_wrapperI3MulmLm4EEvPT0_PmS3_
ctbignum_ctverif.cpp


SMACK program verifier version 1.9.0
SMACK generated a.bpl

Boogie program verifier version 2.3.0.61016, Copyright (c) 2003-2014, Microsoft.

Boogie program verifier finished with 1 verified, 0 errors
```

fast, bug-free, and constant-time

✓ ✓? ✓

# Modulo Operation

# Modulo Operation

- Fixed modulus, known at compile-time
- Barrett / Montgomery reduction (precomputations at compile time)

# Modulo Operation

- Fixed modulus, known at compile-time
- Barrett / Montgomery reduction (precomputations at compile time)
- Montgomery & Granlund: *"Division by Invariant Integers using Multiplication"*, 1994
- Compiler already does this for integers:

```
unsigned long foo(unsigned long a)
{
    return a % 78623419;
}
```

```
foo(unsigned long):                                   # @foo(unsigned long)
        movq    %rdi, %rcx
        movabsq $-2701562103368370257, %rdx # imm = 0xDA821F949A1237AF
        movq    %rcx, %rax
        mulq    %rdx
        shrq    $26, %rdx
        imulq   $78623419, %rdx, %rax   # imm = 0x4AFB2BB
        subq    %rax, %rcx
        movq    %rcx, %rax
        retq
```

Thank you Matt Godbolt!

## Division by Invariant Integers [Granlund—Montgomery94]

- Idea: Let $N$ be an integer multiple of the machine word size, and let $d < 2^N, \quad \ell = \lceil \log_2 d \rceil$

## Division by Invariant Integers [Granlund—Montgomery94]

- Idea: Let $N$ be an integer multiple of the machine word size, and let $d < 2^N$, $\ell = \lceil \log_2 d \rceil$
- Find $m$ such that the following equality holds

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lfloor \frac{m \cdot n}{2^{N+\ell}} \right\rfloor \qquad \forall n : 0 \leq n < 2^N - 1$$

## Division by Invariant Integers [Granlund—Montgomery94]

- Idea: Let $N$ be an integer multiple of the machine word size, and let $d < 2^N$, $\ell = \lceil \log_2 d \rceil$

- Find $m$ such that the following equality holds

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lfloor \frac{m \cdot n}{2^{N+\ell}} \right\rfloor \qquad \forall n : 0 \leq n < 2^N - 1$$

- The remainder is found by performing an additional multiplication and subtraction

## Division by Invariant Integers [Granlund—Montgomery94]

- Idea: Let $N$ be an integer multiple of the machine word size, and let $d < 2^N$, $\ell = \lceil \log_2 d \rceil$
- Find $m$ such that the following equality holds

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lfloor \frac{m \cdot n}{2^{N+\ell}} \right\rfloor \qquad \forall n : 0 \leq n < 2^N - 1$$

- The remainder is found by performing an additional multiplication and subtraction
- Gives constant-time modulo reduction

# Composition With Higher Level Libraries

# Composition With Higher Level Libraries

- E.g., Linear Algebra with eigen http://eigen.tuxfamily.org/

```
using Mx33 = Eigen::Matrix<GF_100_bits_prime, 3, 3>;
Mx33 A, B, C;

A <<  2_Z,   4_Z,  6_Z,
     10_Z,  11_Z, 12_Z,
      1_Z, 100_Z, 30_Z;

B << 5_Z,  3_Z,  9_Z,
     8_Z,  6_Z, 55_Z,
     3_Z, 17_Z,  2_Z;

C = A * B;
```

# Composition With Higher Level Libraries

- E.g., Linear Algebra with eigen http://eigen.tuxfamily.org/

```
using Mx33 = Eigen::Matrix<GF_100_bits_prime, 3, 3>;
Mx33 A, B, C;

A <<  2_Z,   4_Z,  6_Z,
     10_Z,  11_Z, 12_Z,
      1_Z, 100_Z, 30_Z;

B << 5_Z,  3_Z,  9_Z,
     8_Z,  6_Z, 55_Z,
     3_Z, 17_Z,  2_Z;

C = A * B;
```

**...plus some boilerplate...**

```
using GF100_bits_prime =
decltype(Zq(633825300114114700748351602943_Z));

namespace Eigen {
template <>
struct NumTraits<GF100_bits_prime> :
GenericNumTraits<GF100_bits_prime> {
  using Real = GF100_bits_prime;
  using NonInteger = GF100_bits_prime;
  using Literal = GF100_bits_prime;
  using Nested = GF100_bits_prime;
  static inline Real epsilon() { return 0; }
  static inline Real dummy_precision() { return 0; }
  static inline int digits10() { return 0; }
  enum {
    IsComplex = 0, IsInteger = 1, IsSigned = 1,
    RequireInitialization = 1, ReadCost = 1,
    AddCost = 1, MulCost = 1
  };
```

# Key Takeaways

- ctbignum - Library for fixed-size big-int and modular arithmetic: pure Modern C++; features big-integer computations at compile-time
- constexpr functions:
  Compile-time evaluation + competitive runtime performance
  - Use std::integer_sequence for compile-time arrays!
  - However, cumbersome to handle
- Some functions have been formally verified (correctness, constant-timeness)
- Open-source tooling for formal verification of C++ programs has become mature & easy-to-use (SAW, SeaHorn, ct-verif, Smack, ...)
  Give them a try on your C++ code !

# Thank You!

niekbouman@gmail.com

**github.com/niekbouman/ctbignum**

niekbouman.github.io