Why we should care about parallel programming in securing Cyber-physical systems

Sandro Bartolini, Biagio Peccerillo
Department of Information Engineering and Mathematical Sciences
University of Siena, Italy
{bartolini,peccerillo}@dii.unisi.it

Cyber Physical Security Education Workshop - CPSEd
Paris - July 18th, 2017
Outline

• Introduction

• Modular exponentiation parallelization
  • m-ary with precomputation
  • m-ary on-demand
  • Slice method

• Parallel Karatsuba multiplication

• Conclusions
Introduction and motivation – processors

- Processor evolution has changed radically after about 2004
  - (potential) performance continued to scale essentially only through parallelism
  - End-user performance has become harder to extract
Introduction and motivation – *processors 2*

- Nowadays processors are parallel ... more and more parallel
  - Biggest reason was the emerging of *wire-delay issues* ... i.e. *on-chip latency*
  - Also mobile/embedded ones (IoT ... soon?)

Pentium 4 (1)  CoreDuo (2)  i7- 980X (6)  i7-5960X / AMD FX8370 (8)

- GPU and related architectures further push HW parallelism

Kirin 620 smartphone 64-bit ASIC (*8 ARM cores, MALI-450 GPU*)  GTX 1080 (3584 cuda cores)
Introduction and motivation - *interconnects*

- Nowadays we need far more energy to bring operands to the cores (across the chip) than to perform the operation → time to move data around
- For efficiency and performance scalability:

  Elaboration need to be local and parallel!

From Bill Dally’s “GPU Computing to ExaScale and Beyond” keynote, SC’10
Introduction and motivation - *parallelism*

In every field where ‘computational thinking’ is pushed, is nowadays of the utmost importance to promote:

• Parallel programming concepts

In security applications where *performance* and/or *efficiency* is needed, this is particularly appropriate

• Cryptographic algorithms and protocols
• Embedded systems
• Connected systems

\[\text{cyber-physical systems}\]
Introduction and motivation - parallelism

From the educational standpoint it is challenging:

• Parallel architectures are heterogeneous
  – CPUs, GPUs, hybrid ... with different efficient programming strategies and resources

• Parallel programing is complex in itself ... and debugging is worse 😊
  – Imperative programming is implicitly sequential
  – proving specific techniques are needed

• Big interaction with computer-architecture
  – caches, coherence, memory consistency model
  – Hyperthreading, processor microarchitecture

• Big interaction with operating system
  – Thread orchestration and management
  – Scheduling, migration, etc
Introduction and motivation – parallelism (2)

Need to promote awareness around parallel programming in the security domain
• Very crucial as cryptographic algorithms were devised without parallelism in mind
• Also from the mathematical standpoint, most of the primitives are intrinsically sequential
  – Maybe it needs to be like this for security reasons?

We will address two fundamental algorithms of cryptography
• Modular exponentiation (as in RSA)
• Multiplication of big numbers

We propose and discuss a few parallelization strategies
• Educational approach to highlight phenomena without looking for the ultimate performance/optimizations
Outline

• Introduction

• Modular exponentiation parallelization
  • m-ary with precomputation
  • m-ary on-demand
  • Slice method

• Parallel Karatsuba multiplication

• Conclusions
Modular exponentiation - intro

Modular exponentiation: $M^e \mod(n)$

- With $M$, $n$ and possibly $e$ being ‘big’ enough for security (k-bits)
  - E.g. in current RSA 2048- to 4096-bit are deemed safe in the short term
- Square-and multiply or binary method
  - Given the binary expansion of $e = (e_{k-1}, e_{k-2}, \ldots, e_1, e_0)$

$$e = (e_{k-1}e_{k-2}\cdots e_1e_0) = \sum_{i=0}^{k-1} e_i2^i$$

for $e_i \in \{0, 1\}$. The binary method for computing $C = M^e \mod(n)$ is given below:

**The Binary Method**

*Input: $M, e, n$. 
Output: $C = M^e \mod n$. 

1. if $e_{k-1} = 1$ then $C := M$ else $C := 1$
2. for $i = k - 2$ downto 0
   2a. $C := C \cdot C \mod n$
   2b. if $e_i = 1$ then $C := C \cdot M \mod n$
3. return $C$

* From: Koç, Ç. K. «High-Speed RSA Implementation», RSA Laboratories, 1994
m-ary approach with precomputation - intro

The exponent can also be scanned also \( \log_2(m) \)-bits at a time \( \rightarrow m\text{-ary method} \rightarrow \) reduces number of modular multiplications

At each step:

- \( \log_2(m) = r \) squarings need to be done on the operand
- Then a multiplication by a specific power of the base
  - The powers needed are \( M^2, M^3 \ldots, M^{m-2}, M^{m-1} \)
  - E.g. 3-ary \( \rightarrow \) powers needed 1 (trivial), 2, 3, 4, 5, 6, 7
- which can be pre-computed before the scan \( \rightarrow \) precomputation table

Parallel approach:

- Before exponent scan, \( N_p \) threads prepare the precomp-table
  - Powers evenly split between the threads ... simple, can be improved!
- Split the exponent in \( r \)-bit slices and we group them in a «comb-like» fashion
- E.g. number of working threads \( N_t = 4 \)
  - every \( r \)-bit slice of the exponent whose index mod(\( N_t \)) is 0 \( \rightarrow \) thread 0
  - every \( r \)-bit slice of the exponent whose index mod(\( N_t \)) is 1 \( \rightarrow \) thread 1
  - ...
  - every \( r \)-bit slice of the exponent whose index mod(\( N_t \)) is \( N_t - 1 \) \( \rightarrow \) thread \( N_t - 1 \)
m-ary approach with precomputation – intro (2)

Each thread performs a reduced amount of multiplications

Work of each thread is quite balanced ($e$ is thousands bits, $r$ a few bits)

Work execution time is limited by the exponentiation by $e_3$ ...
m-ary with precomputation - results

Experiments run on:

- Dual Xeon E5-2650 v2 @2.60GHz (3.4 GHz turbo), 8c/16t, L1pc 32K+32KB, L2pc 256KB, L3sc 20 MB, 64 GB RAM

- **Enforced thread-to-core affinity**: big difference in the results
  - Thread-data resources are not negligible and occupy cache space
  - Thread can be migrated by the OS → unnecessary cold misses
  - Same physical processor

- Linux Debian 8 or 9 operating system
- Key sizes: 1024, 2048, 4096, 8192, 16384, 32768
- Repeated experiments:
  - From 10s to 1000s times to let benchmark run from 10s of seconds up to a few minutes for every key size
- Benchmarks implemented in C++ relying on GMP/MPIR libraries

- Showing improvements over plain square-and-multiply (S&M)
  - GMP/MPIR native performance is shown as a reference
m-ary with precomputation – results 1024 (1 thr)

- 1-thread with 1/2 pre-comp threads: always worse than S&M
  - Very slow for 4-ary or 6-ary versions
- 1-thread with 4 pre-comp threads (4-pt): -9% (2²-ary) and -6% (2³-ary)
m-ary with precomputation – results 1024 (4-thr)

- 4-threads:
  - from -15% to -20% (2²-ary, 1-4 threads)
  - -23% (2³-ary, 4 pre-threads), -20% (2²-ary, 2-/4-pt), [-18% : -15%] (2²-ary, 1-pt; 2³-ary, 2-pt; 2⁴-ary, 4-pt)
m-ary with precomputation – results 1024 (8-thr)

- **8-threads:**
  - Best configuration: -26% (\(2^4\)-ary, 4 pre-computation threads)
  - 8 configurations (m-ary, pt) -17% or better improvement
m-ary with precomputation – results 2048

- 1-thread: need 4 pre-comp threads to get -14% on $2^2$-ary
- 4-threads: higher $2^r$-ary configurations sustainable only with multiple pre-comp threads:
  - -29.5% ($2^2$-ary, 4-pt), -25% ($2^3$-ary, 4-pt; $2^4$-ary, 4-pt; $2^2$-ary, 2-pt)
- 8-threads: -25% (various $2^r$-ary, n-pc configurations)
m-ary with precomputation – results 4096

- Increasing key size is easier to exploit pre-computations
  - More configurations get advantages
  - 1-thread: -10% \((2^2\text{-ary}, 4\text{-pc})\)
  - 4-threads: -23% \((2^3\text{-ary}, 4\text{-pc})\)
  - 8-threads: -26% \((2^3\text{-ary, 4-pc})\)
m-ary with precomputation – results 8192

- Main threads are needed to take advantage of pre-computations
  - 4 or 8 are similar
- 2/4 pre-computation threads are needed to exploit m-ary (even from $2^2$-ary)
- Best performance: **-23.5% (8-threads, $2^2$-ary, 4-pt)**
m-ary with precomputation – results 16k & 32k

• Similar results: Stability of the approach across key sizes

• Best performance:
  – 16384: -24% (8-threads, 2^{3}-ary, 4-pt)
  – 32768: -24.5% (8-threads, 2^{3}-ary, 4-pt)
m-ary with precomputation: wrap up

• 2/4 pre-computation threads can improve m-ary performance
  – Up to -25% / -30% improvement

• m-ariety from $2^2$, $2^3$ typically gives best results
  – 8-threads and 4-pt can exploit
    • $2^4$-ary computation (average improvement: -22%, max -26%)
    • $2^6$-ary computation (average improvement: -14%, max -20%)

• Problems:
  – Pre-computations are performed before starting the computation
  – Pre-computed values are global
    • Cache management can add overhead in the (first) thread access to the values
    • Cache hierarchy traversal
  – Big $m$: not all precomputations are statistically used
Outline

• Introduction

• Modular exponentiation parallelization
  • m-ary with precomputation
  • m-ary on-demand
  • Slice method

• Parallel Karatsuba multiplication

• Conclusions
m-ary “on-demand” - intro

N-threads are started immediately
• Each one doing the same work as in the «M-ary with precomputation» method

• Every time a thread looks for a precomputed value and finds it not available:
  – Locks the precomputation table entry
    • A first attempt locked the whole table → no concurrency in precomputations, especially in the early stages
  – Calculates the needed power
  – Fills the table entry
  – Unlocks the table entry

• Pros:
  – Computation starts immediately
  – Only the required precomputed entries are calculated
    • Useful for bigger $m$-ary approaches
  – Still cache hierarchy traversal for getting entries where needed
m-ary “on-demand” – results 1024

- More configurations improve, compared to the preliminary pre-computation case
- 1 thread exposes the effect of m-ary approach: best at 2⁴-ary (-22%)
- Increasing thread number is beneficial especially for bigger tables (2⁸/2¹⁰-ary)
  - Sort of saturation at 2/3 threads for 2⁴-ary
  - Sweet spot at 3-threads 2³-ary (-32%)
m-ary “on-demand” – results 4k & 8k

- Bigger keys benefit from bigger tables
  - On-demand approach limits useless work
- Increasing thread number is beneficial especially for bigger tables (8/10-ary)
- Best configurations
  - 4096: -27% (2^6-ary, 3 threads), -27% (2^4-ary, 4 threads), -26% (2^4-ary, 6/7/8 threads)
  - 8192: -27% (2^4/2^6-ary, 3 threads), -25% (2^3/2^4-ary, 6/7/8 threads) -25% (2^6-ary, 3 threads)
m-ary “on-demand” – results 16k & 32k

- Pre-computation tables can be exploited also by a few threads
  - Less benefits from increasing beyond 5 threads

- Best configurations
  - 16384: -27% (2⁶/2⁸-ary, 3 threads), -26% (2⁶-ary, 7 threads)
  - 32768: -27% (2⁶-ary, 2/8 threads), -26% (2⁸-ary, 7/8 threads)
m-ary “on-demand”: wrap up

• Solution quite robust in the number of threads needed
  – 3-threads or 6/7/8 threads are the best configuration
  – Up to -32% (1024) and never less than -27% in the other cases
    • Various «ariety» possible and beneficial: also $2^6$-$2^8$-ary

• Problems:
  – Possible conflicts between threads at small «ariety» when the same pre-computation is needed
    • Amortized for bigger keys and less likely for bigger «ariety»
  – Pre-computed values are global
    • Cache management can add overhead in the (first) thread access to the values
  – Big $m$: not all precomputations are statistically used
Outline

• Introduction

• Modular exponentiation parallelization
  • m-ary with precomputation
  • m-ary on-demand
  • Slice method

• Parallel Karatsuba multiplication

• Conclusions
Slicing - intro

N-threads are started:

- Each one gets assigned a contiguous «slice» of the exponent
  - The other lower bits are zeroed
- After all complete the work: sub-results are multiplied together

\[ M^e \mod(n) = R0 \times R1 \times R2 \mod(n) \]
Slicing – intro (2)

• Cons:
  – The load of the threads is quite unbalanced
  – Their overall computation time is bounded by the one with the most significant slice
    • After the «slice» exponentiation each thread performs a chain of modular squares (apart from the first slice)

• Pro:
  – The load of the more significant slices can be made thinner with uneven exponent slicing
  – Optimally balanced approaches have been proposed [1]
  – Sequences of squares can be cache-friendly: both data and instruction

Slicing – results

- 3/4 threads, and slices, are typically enough to get the maximum benefit
- Smaller key sizes are accelerated more
  - Up to -40% for 1024-bit (4-threads/slices)
  - Up to -36.5% for 2048-bit (10-threads/slices), -35.5% (6-thread/slices)
- From 4096 and up, speedup reaches -30% at 3/4 threads/slices
- ‘Optimum’ slicing does not have measurable effect
Slicing: wrap up

- 3/4 threads/slices, are enough to get the maximum benefit
  - More threads do not alter performance

- Speedups:
  - Up to -40% for 1024-bit, -36.5% for 2048-bit, -30% for 4096-32768-bit

Observation:

- Fastest, and stable, even if threads manage unbalanced work, why?

- The unbalanced work is *simple* and *repetitive*
  - modular squaring
  - *Simple*: not involving big data structures and simpler than modular multiplication
    - Small memory footprint $\rightarrow$ L1 / L2 caches can support the execution
  - *Repetitive*: many squaring needed in a row
    - Temporal locality $\rightarrow$ compiler+processor+cache can support fast execution
Outline

• Introduction
• Modular exponentiation parallelization
  • m-ary with precomputation
  • m-ary on-demand
  • Slice method
• Parallel Karatsuba multiplication
• Conclusions
Parallel Karatsuba - intro

A number of crypto-algorithms rely on modular multiplication of big numbers

Karatsuba algorithm (1960) is a multiplication algorithm that
• Reduces the asymptotic complexity of multiplication from \(O(n^2)\) to \(O(n^{1.583})\)
• Relies on a ‘divide-and-impera’ approach
• The multiplication of the \(x\) and \(y\) (N-bits each) can be done considering the two ‘halves’ of each number \(x = x_1 \cdot 2^{\frac{N}{2}} + x_0, y = y_1 \cdot 2^{\frac{N}{2}} + y_0\)

\[
\begin{array}{c|c}
\text{high} & \text{low} \\
\hline
x_1 & x_0 \\
y_1 & y_0
\end{array}
\]

• And doing three \(N/2\) bits multiplications

We have implemented a parallel version with 3-threads (the main one, plus two auxiliary ones)
• Each thread perform a \(N/2\)-bit multiplication
• and after all are done, the main thread composes the final result
Parallel Karatsuba – implementation (sequential)

For investigating parallelism speedup we implemented a sequential karatsuba as a reference

• Same data structures and same management as the parallel ones
• Note: using C++14 here ...

```cpp
mpz_class karaMul(mpz_class const& x1, mpz_class const& x2)
{
  assert(x1.get_mpz_t()->_mp_size == x1.get_mpz_t()->_mp_size); // per org
  auto const part1 = splitBigNum_limb(x1); // part1 is { high_bits1, low_bits1}
  auto const part2 = splitBigNum_limb(x2); // part2 is { high_bits2, low_bits2}

  mpz_class x1Lx2L = part1.second*part2.second; // multiplication 1
  mpz_class x1Hx2H = part1.first*part2.first; // multiplication 2
  mpz_class midTerm = x1Hx2H + x1Lx2L - (part1.first-part1.second) * (part2.first-part2.second); // multiplication 3

  mp_bitcnt_t halfBits = (x1.get_mpz_t()->_mp_size/2) * sizeof(mp_limb_t) * 8;
  mpz_class ret = x1Lx2L + (midTerm << halfBits) + (x1Hx2H << (2*halfBits));

  return ret;
}
```
Parallel Karatsuba – implementation (async)

std::async are C++ standard task wrappers which
• Can execute a function in a separate thread
• Return a handle to the result (std::future) for the caller
• The caller can block on the future waiting for the result
• Quite high-level and simple to use → overhead?

```cpp
mp_class karaMulThrAs(mp_class const& x1, mp_class const& x2)
{
    mp_bitcnt_t halfBits = (x1.get_mp_t()->mp_size/2) * sizeof(mp_limb_t) * 8;
    auto const part1 = splitBigNum_limb(x1);
    auto const part2 = splitBigNum_limb(x2);
    auto retLL = std::async(std::launch::async, standardMul, part1.second, part2.second);
    auto retHH = std::async(std::launch::async, standardMul, part1.first, part2.first);
    mp_class midTerm = -(part1.first-part1.second) * (part2.first-part2.second);
    mp_class x1Lx2L = retLL.get();
    mp_class x1Hx2H = retHH.get();
    midTerm += x1Lx2L + x1Hx2H;
    mp_class ret = x1Lx2L + (midTerm << halfBits) + (x1Hx2H << (2*halfBits));
    return ret;
}
```
Parallel Karatsuba – implementation (threads)

std::threads are C++ standard thread handles which
• Can execute a function in a separate thread
• Are lower-level than std::asyncs
• We need to explicitly manage the synchronization for getting the result.
  – Specifically, joining thread execution explicitly

```cpp
mpz_class karaMulThr(mpz_class const& x1, mpz_class const& x2)
{
    mp_bitcnt_t halfBits = (x1.get_mpz_t()->mp_size/2) * sizeof(mp_limb_t) * 8;
    auto const part1 = splitBigNum_limb(x1);
    auto const part2 = splitBigNum_limb(x2);

    mpz_class x1Lx2L;
    auto thr1 = std::thread(mulThr, part1.second, part2.second, std::ref(x1Lx2L));
    mpz_class x1Hx2H;
    auto thr2 = std::thread(mulThr, part1.first, part2.first, std::ref(x1Hx2H));

    mpz_class midTerm = -(part1.first-part1.second) * (part2.first-part2.second);
    thr1.join();
    thr2.join();
    midTerm += x1Lx2L + x1Hx2H;

    mpz_class ret = x1Lx2L + (midTerm << halfBits) + (x1Hx2H << (2*halfBits));
    return ret;
}
```
Parallel Karatsuba – implementation (threads) discussion

A problem can be that multiplication algorithm is pretty fast

<table>
<thead>
<tr>
<th>key_size [bits]</th>
<th>GMPmul [us]</th>
<th>Kara_seq [us]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.61584</td>
<td>1.72731</td>
</tr>
<tr>
<td>2048</td>
<td>0.873759</td>
<td>2.29119</td>
</tr>
<tr>
<td>4096</td>
<td>2.57911</td>
<td>4.07022</td>
</tr>
<tr>
<td>8192</td>
<td>7.86512</td>
<td>9.55777</td>
</tr>
<tr>
<td>16384</td>
<td>21.1608</td>
<td>26.396</td>
</tr>
<tr>
<td>32768</td>
<td>55.902</td>
<td>67.8586</td>
</tr>
<tr>
<td>65536</td>
<td>153.327</td>
<td>174.204</td>
</tr>
</tbody>
</table>

... compared to the thread spawn and spawn+join time:
- i7 2600: 6.5 us / 16.0 us
- E5-2650 v2: 4.3 us / 10.1 us
- i7 6800K: 4.4 us / 10.4 us
Parallel Karatsuba – implementation (thread pool)

Spawnning and releasing resources of new threads can be quite costly compared to the time to perform a multiplication on big integers

- The ‘thread-pool’ solution:
  - The helping threads are always ‘active’ and are waiting on a condition-variable (CV) within an infinite loop
  - The main thread fills the threads’ input structures with the operands and triggers their awakening
  - They compute the multiplication, store the result in a data structure accessible from the main thread and block again
  - Once the main thread wants a result, it checks the result’s CV and either blocks waiting, or proceeds and is able to use the result value

Pros: no overhead from thread spawn/release
Cons: more complex parallel solution
Parallel Karatsuba – implementation (thread pool) - 2

Problem: multiplication is fast ... compared to wake up a thread

- thread activation time [us] (cond.variables+mutex)
  - different cores, threads to be unblocked
  - they lock on their own after their computation is done (no ‘join’ equivalent)
    - i7 2600: 4.1 us
    - i7 6800K: 2.3 us
- thread activation time [us] (cond.variables+mutex)
  - same core as main thread, threads to be unblocked
    - i7 2600: 1.6 us (interesting! but not usable in this case)
    - i7 6800K: 1.3 us

- Cons: condition variable wait and unlock still induces a non-negligible overhead
Parallel Karatsuba – implementation (thread pool) - 3

Improvement of infinite-thread: **lock-free data structures**

- between main thread and helper ones, for feeding operands
- Between helper threads and main one, for retrieving results
- Without explicit synchronization mechanism between the two
  - No OS intervention
  - No scheduler
  - No allocation of data structures or thread resources

- thread activation time [us] (cond. variables+mutex)
  - different cores, threads to be unblocked
  - they lock on their own after their computation is done (no ‘join’ equivalent)
    - i7 2600: 0.12 us (interesting !)
    - i7 6800K: 0.18 us

- Then, in any case operands need to be «prepared» in a suitable way to be fed to the helper threads
Parallel karatsuba – results methodology

Experiments run on:
- i7-6800K @ 3.40GHz (3.6 GHz turbo), 6c/12t, L1pc 32K+32KB, L2pc 256KB, L3sc 15 MB, 128 GB RAM
- Dual Xeon E5-2650 v2 @2.60GHz (3.4 GHz turbo), 8c/16t, L1pc 32K+32KB, L2pc 256KB, L3sc 20 MB, 64 GB RAM
- i7-2600 @3.4 GHz (3.8GHz turbo), 4c/8t, L1pc 32K+32KB, L2pc 256KB, L3sc 8 MB, 32 GB RAM
- **Enforced thread-to-core affinity**: not big issue in this case
- Linux Debian 8 or 9 operating system
- Key sizes: 1024, 2048, 4096, 8192, 16384, 32768, 65536
- Repeated experiments:
  - From 5000s to 30000s times to let benchmark run for a reasonable amount of time for every key size
- Benchmarks implemented in C++ relying and using GMP/MPIR library for Big numbers and reference
- Showing improvements over plain sequential Karatsuba and GMP/MPIR
Parallel Karatsuba – results parallel simple

- `std::sync` and `std::thread` approaches are worse/equal than sequential up to 16384
  - 32768: -35% (std::async), -37% (std::thread)
  - 65536: -52% (std::async), -48% (std::thread)
  - **Spawn/join threads overhead is limiting the approach**

- **Improvements over GMPmul for big numbers**
  - 32768: -21% (std::async), -25% (std::thread)
  - 65536: -45% (std::async), -41% (std::thread)
Parallel Karatsuba – results parallel inf-threads

- Infinite-threads improve over simple parallel tasks
  - Where they were already good
  - -37% than Kara_seq @16384 (-21% than GMP)
  - Matches Kara_seq @8192
  - For smaller keys: threads synchronization overhead and parameter passing is limiting the approach
Parallel Karatsuba – results parallel inf-threadsLF

- Lock-free infinite-threads improve over infinite-threads
  - Number size decreases → improved advantage
  - Better for small keys: -46% @8192 than Kara_seq (-35% vs GMP)
  - -25% @4096 than Kara_seq
  - -3% @2048 than Kara_seq
  - threads parameter preparation and passing is limiting the approach
Parallel Karatsuba – results parallel inf-threadsLF (2)

- Normalized results:
  - 4096: -24% vs Kara_seq, +17% vs GMP
  - 8192: -45% vs Kara_seq, -33% vs GMP
  - 16384: -54% vs Kara_seq, -41% vs GMP
  - 32768: -58% vs Kara_seq, -49% vs GMP
  - 65536: -61% vs Kara_seq, -55% vs GMP
Parallel Karatsuba – results parallel inf-threads LF (3)

i7-6800K @ 3.40GHz (3.6 GHz turbo), 6c/12t, L1pc 32K+32KB, L2pc 256KB, L3sc 15 MB

- newer HW: slightly better performance on sequential code ↦ turbo frequency, ILP
  - 4096: -15% vs Kara_seq, +49% vs GMP
  - 8192: -38% vs Kara_seq, -19% vs GMP
  - 16384: -52% vs Kara_seq, -39% vs GMP
  - 32768: -56% vs Kara_seq, -48% vs GMP
  - 65536: -61% vs Kara_seq, -54% vs GMP
Parallel Karatsuba: wrap up

• Only 3 overall threads can significantly speed up multiplication
  – Over plain Karatsuba sequential from 4096 keys (-15%/-25%) up to -55%/60%
  – Over GMP: less than -30% @8192 and up to less than -50% for bigger cases
  – higher-number of threads could help, especially for bigger keys

• Parallelization opportunities and hurdles
  – Difficulty of programming: abstractions → overhead vs abstraction
  – Overhead of thread work orchestration
  – Interactions with Operating System (OS) and computer architecture (caches, etc)
  – Lack of HW + parallel programming support at the μs scale

• Some parallel-programming strategies can be plug-in to fit the problem
  – Thread-pool and lock-free techniques → complexity
Outline

• Introduction

• Modular exponentiation parallelization
  • m-ary with precomputation
  • m-ary on-demand
  • Slice method

• Parallel Karatsuba multiplication

• Conclusions
Conclusions

• Exponentiation, as it is, can be accelerated through parallelism
  – -30/-40%

• Big number multiplication, as it is, can be accelerated through multi-threaded implementations
  – -15% to -60% vs sequential Square&Multiply
  – -19% to -54% vs native GMP/MPIR

Good case for promoting education into parallel programming in general and, specifically, in the cyber-physical system security
  – Parallelism in hardware is not emerging ... is already happened at almost all levels, from embedded to HPC
  – Need to harness it and exploit it now!

Discussion:

• Different math algorithms could be devised to be more parallelism-friendly?
Why we should care about parallel programming in securing Cyber-physical systems

Thanks for your attention!

Q & A

Cyber Phisical Security Education Workshop - CPSEd
Paris - July 18th, 2017

Sandro Bartolini, Biagio Peccerillo
Department of Information Engineering and Mathematical Sciences University of Siena, Italy
{bartolini,peccerillo}@dii.unisi.it