A Benchmarking Framework for Zero-Knowledge Proof Systems

3rd ZKProof Workshop
May, 2020

Daniel Benarroch
QEDIT

Aurélien Nicolas
QEDIT

Justin Thaler
Georgetown University

Eran Tromer
Columbia and TAU

Why we need a benchmarking framework?

Many schemes

- Many zk proof systems available.
- Built w/ different applications in mind.
- No one is best.
- Deluge of new ones; nonstop innovation

"Apples-to-Apples" - an ideal

- Dozens of costs/properties of a proof system are relevant in applications.
- Designers emphasize their preferred tradeoff in comparisons.
- Hard to get accurate picture of all costs

Why is it difficult to achieve?

Evaluating/comparing proof systems is subtle.

Complicated tradeoffs of costs and features

Performance depends heavily on

Modes of operation (batching, amortization, helpers, etc...)

Types of statements

Sizes of statements

Security level and assumptions

Optimization efforts

Barriers to continued development / adoption

- Difficult to evaluate progress in the area.
- Challenging for developers to identify the proof system best suited to their needs.

Goals of this Proposal

- Carefully enumerate challenges and subtleties in benchmarking proof systems
- Articulate best practices for benchmarking and reporting
- Discuss tooling to aid in benchmarking
- Put forward a concrete framework for benchmarking a proof system
    - i.e., a list of functionalities to test the proof system on, and a list of costs and properties/features that should be reported.

Some important concerns with the goals

Avoid stifling innovation

Rigid or narrow framework will impose artificial constraints on protocol designers

Avoid biases

Inadvertently favoring some approaches over others

How do we evaluate and compare proving systems?

Quantitative Costs

- Prover time
- Prover space
- Verifier time
- Verifier space
- Proof size
- Size of public keys and parameters
- Rounds of interaction
- Security level (statistical or computational)

Qualitative Costs

- Hardness assumptions (comp vs. PQ)
- Setup assumptions (SRS, URS; universal / specific; updateable?)
- Zero-knowledge (statistical vs comp.)
- Simplicity and ease of verifying correctness
- Parallelization and acceleration
  - Can we parallelize or distribute the prover's computation?

More challenges in developing a benchmarking framework

Main Challenges

Identifying functionalities

Identify "representative" functionalities to benchmark

Design flexibility

Specificity, optimizations, frontend vs. backend

Measurement and accounting

Fixed input sizes vs. scalability, accounting

Underlying environment

Hardware-dependent costs

1/ Identifying functionalities: complications

MANY STATEMENTS OF INTEREST

- Cryptographic vs non-cryptographic
- Simple (range proofs) vs complex (software properties, arbitrary data analysis tasks)
- Lots of avenues for innovation; avoid prescribing specific design paradigm

LARGE TRADEOFF SPACE

- Speed vs scalability
- Complicated, multidimensional cost tradeoffs common

DATA UNIVERSE STRUCTURE

- Idiosyncratic inputs in some applications/contexts

1/ Identifying functionalities: a high-level solution

CHALLENGE
Choose a representative set of functionalities

Useful in some existing applications

Dominant (bottle-neck) cost in applications

Diversification

Main Challenges

Identifying functionalities

Identify "representative" functionalities to benchmark

Design flexibility

Specificity, optimizations, frontend vs. backend

Measurement and accounting

Fixed input sizes vs. scalability, accounting

Underlying environment

Hardware-dependent costs

2/ Designer Flexibility: Hash Function Pre-image

Does protocol designer get to choose the hash function?

YES

- Quantify security
- Sufficient effort on cryptanalysis?
- How much efficiency due to ingenuity in hash function vs. proof system?

NO

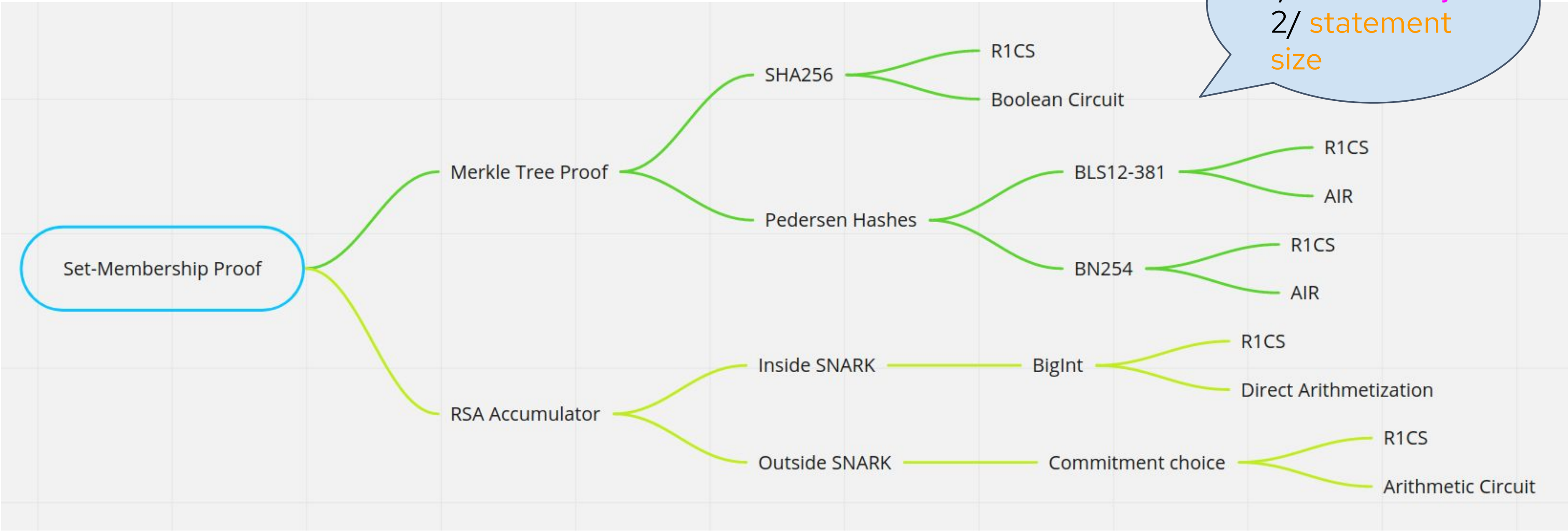
Not capturing goal of developing most efficient protocol for the general functionality.

Possible Solution: define functionalities at varying levels of specificity

Example Benchmark Functionality

Example of branching: Set-Membership

branchout:
1/ bit security
2/ statement size



Enabling flexibility of choice

1. High-level: Functionality	Full design flexibility; Optimize statement for proof machinery
2. Intermediate-level(s): Instantiation of functionality	Optimize front-end (encoding method) & proof machinery for given instantiation
3. Lower-level: Fixed IR Language (e.g., R1CS, arithmetic circuit)	Optimize front-end & back-end for given IR language
4. Bottom-level: Fixed IR statement	Optimize the back-end (proof machinery)

Important Considerations (Part 1)

Overfitting

- Hard to prevent people from optimizing for specific benchmark functionality
- Danger of inflating importance of bottom-level

- Different levels allow people to better understand the efficiencies and inefficiencies of their schemes
  - Could reveal more about the source of each
- Too many branches to keep track of?
- Designers should be able to submit functionalities?

Important Considerations (Part 2)

- Cryptographic functionalities often of interest as "subroutines" in larger protocols.
  - Example: Set-membership used in SNARKs for RAM execution to enforce memory consistency.
  - Structured data may be different from application to application
- Set-membership benchmark may not assume any structure.
  - Ignoring structure may mean overheads (serializing field elements into bits)

The structure is highly application-dependent, what should we do?

Main Challenges

Identifying functionalities

Identify "representative" functionalities to benchmark

Design flexibility

Specificity, optimizations, frontend vs. backend

Measurement and accounting

Fixed input sizes vs. scalability, accounting

Underlying environment

Hardware-dependent costs

3/ Measurement and accounting

Literature is rife with ambiguity.

What counts toward:

Verifier time or space?

Input processing (e.g., reading input, LDE evaluation)	Key (CRS) Generation	Reading the CRS	Proof verification

Prover time or space?

Native Execution	Witness Reduction	Other Pre-Processing	Proof generation

Main Challenges

Identifying functionalities

Identify "representative" functionalities to benchmark

Design flexibility

Specificity, optimizations, frontend vs. backend

Measurement and accounting

Fixed input sizes vs. scalability, accounting

Underlying environment

Hardware-dependent costs

4/ Underlying environment (Part 1)

All hardware favors some operations over others

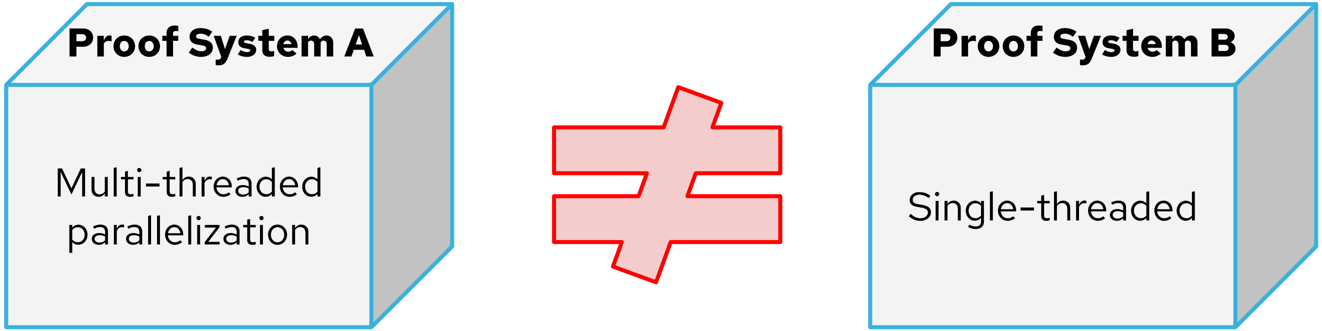
Critical to have portable and modifiable implementations

Machines with massive memory are more forgiving to memory-hungry back-ends

4/ Underlying environment (Part 2)

Benchmarking parallelization is subtle.

Different protocols are more or less amenable to different parallel hardware (GPUs vs. clusters)



Recommendation: Reported benchmarks should include single-threaded CPU performance on a precisely specified machine, to establish baseline runtime that can be reproduced and compared.

Tooling and Analysis

Measurement method

Running a benchmark

- Simply run a command + parameters
- Standard functionalities (high-level and low-level):

```
{ function: "set_membership",
  set_size: 65536 }
{ function: "r1cs" }
```

+ Standard encoded circuit.

Reporting

- Self-report metrics because implementers know better (ignore startup, estimate memory).
- Simply `print` in a standard format:

```
ZKPROOF_BENCHMARK:
{ proving_time: …, max_memory:… }
```

Process and publication

- Curate a list of test commands
- Accumulate measurements into a global results file
- Aggregate results into tables and graphs
- Publish regularly on github / zkproof.org

\+ **Develop tools for these tasks**

Discussion Points

- How do we NOT stifle innovation?
    - Is such a framework identifying a small number of “representative” functionalities desirable?
    - At least a standard way of presenting the benchmark in question / best-practices / guidelines to reduce biases
- Functionalities and design flexibility
    - How do we prevent (or encourage) designers from (not) tailoring solutions to benchmark functionalities
    - Are any levels more important that others, and do they represent the full spectrum of design choices?
    - What are the functionalities we should choose?
- The structure of data is highly application-dependent, what should we do?

Thank you!