Community Proposal: A Benchmarking Framework for (Zero-Knowledge) Proof Systems

Daniel Benarroch[*], Aurélien Nicolas[*], Justin Thaler[*,+], and Eran Tromer[*,3,4]

[*]QEDIT
[+]Georgetown University
[3]Columbia University
[4]Tel Aviv Univeristy

April 9, 2020

Abstract

This document proposes a partial framework for evaluating the concrete performance of proof and argument systems. The goals of this work are to: (1) summarize the challenges and subtleties inherent in any evaluation framework, (2) encourage quality and consistency in published evaluations, and (3) ease comparison of different proof and argument systems.

Contents

1 Introduction

There is a large and ever-increasing number of (zero-knowledge) proof systems available, and at least a dozen different costs or properties of a proof system that may be relevant in various applications (see Section 1.1). Currently, there is no one best scheme, and it seems likely that there never will be (i.e., certain tradeoffs between costs may be inherent in any zero-knowledge proof system).

Comparing the strengths and weaknesses of any two proof systems is typically a subtle endeavor, involving complicated tradeoffs that may themselves heavily depend on a variety of factors, including the kinds of statements to be proven, the sizes of the statements, the desired security level, etc.

The above factors make it quite difficult to compare different proof systems, or to evaluate the importance or effectiveness of a new proposal. Consequently, it can be hard to evaluate progress in this research area, and challenging for developers (who understand the demands of a particular application) to identify the proof system best suited to their needs. These are severe barriers to the continued development and adoption of zero-knowledge proofs.

Goals. The goal of this proposal is to address these barriers by taking steps toward a common framework and recommended practices for benchmarking proof systems.

A variety of challenges and subtleties arise when creating such a benchmarking framework. Many have no easy answers. This document attempts to carefully delineate these subtleties, and arrive at a proposal to address each one.

A principal consideration throughout is to avoid stifling innovation. An overly rigid or narrow framework runs the risk of imposing unnecessary constraints on protocol designers, and may inadvertently favor some approaches over others.

Relation to the ZKProof Community Reference. In the following, we rely extensively on the framing and terminology provided by the ZKProof Community Reference [com19]. The Community Reference also contains a brief discussion of challenges and best practices for benchmarking [com19, Section 3.5]. The present proposal aims to continue that work, by providing a deeper and more concrete treatment of the benchmarking process for proof systems. The intent is to replace the aforementioned section by a brief overview and a reference to the present proposal.

1.1 Measuring Proof Systems

There are numerous axes on which to concretely compare zero-knowledge proof systems. The following factors contribute to the large number of relevant axes for comparison:

Proof systems have several components: Most essentially, the prover and verifier. Some schemes also have a generator for setting up common parameters. Some schemes allow the prover and/or verifier to be separated into a pre-processing portion (i.e., which can be executed before the instance, witness, or proof are known) and an online portion.[1]

Different proof systems may have different modes of operation: Some proof systems are targeted at settings involving batching, aggregation, or amortization: of proofs by the same prover, of

[1]For structured (uniform) computations, some proof systems are capable of having the verifier run in time sublinear in the size of the computation, even without any pre-processing phase. Others require a pre-processing phase as expensive as the computation being outsourced, regardless of whether that computation is structured/uniform.

proofs by different provers, or of verification. Some proof systems enable recursive composition or Proof-Carrying Data with various topologies.

Cost metrics: Each component of a proof system incurs many relevant costs, including: running times; memory consumption; proof sizes; the size of public keys and parameters, and rounds of interaction. Also of interest are the counts of low-level operations such as field operations, group operations or (parameterized) Fourier Transforms.[2]

Quantifiable security: Concrete proof systems have a certain security level, whether statistical or computational. In the latter case, the security level captures the cost necessary to produce a false proof, and this might be measured in various ways involving time, memory and/or invocation of idealized primitives; moreover it may involve computational assumptions with their own parameters.

1.2 Qualitative Properties

ZK proof systems are also characterized by numerous qualitative properties. These are outside the scope of the current proposal, but we mention them here for context and as opportunities for further quantitative treatment.

Computational assumptions and models. What assumptions are made? What is known about their plausibility? Are they computational or combinatorial (e.g., coding theory)? Is the scheme plausibly post-quantum secure? What non-standard models are used (e.g., random oracle, generic group, algebraic group)? Are there informal heuristics applied (e.g., Fiat-Shamir deinteractivization without a tight analysis, or instantiation or random oracles) and what is known about their security?

Setup. Does the scheme require a Uniform Reference String (implicitly or explicitly)? Does it require a Structured Reference String, and if so, is it circuit-specific, completely universal, or universal for all circuits of a given size? Is the scheme resistant to subversion of the setup, and in which sense?

Verifiability model and strength. Are proofs publicly-verifiable or designated verifier? Does the scheme provide plain soundness, or knowledge-soundness (proof of knowledge)? Is it statistical or computational[3] (see above for security level)? Are proofs non-malleable, simulation-sound, simulation-extractable?

Zero-knowledge and its strength. Are these proofs zero-knowledge[4], and if so is it statistical or computational (see above for security level)?

Simplicity. How easy is it to understand, implement and verify the scheme?

Parallelization and acceleration. Is the scheme amendable to parallelization, distributed execution, hardware acceleration via GPUs or ASICs, etc.? (This may be qualitatively reflected by specific implementations.)

[2]These are useful for platform-agnostic comparisons, or for estimating running times on new platforms or using hardware acceleration).

[3]Computationally-sound proof systems are also called *arguments*. We follow the ZKProof convention of using the term ”proof system” to denote both statistically-sound and comptuationally-sound ones.

[4]Proof systems may be used for their succinctness property (i.e., proofs may be smaller and easier to verify than the full witness), even if they are not zero knowledge.

1.3 Additional Considerations

In this section, we provide relevant context on how proof systems work, and delineate a large number of subtleties and challenges that any benchmarking proposal must address. In Section 2, we summarize our proposed solutions to these challenges.

Front Ends and Back Ends. Implementations of general-purpose zero-knowledge proofs typically use the "front end + back end" paradigm [ZKP19, Sections 3.2 and 3.3]. In this paradigm, the computation that the prover is supposed to verifiably execute is first expressed in some *intermediate representation (IR)*, typically a Boolean or arithmetic circuit, or a rank-1 constraint system (R1CS). In general, the term IR refers to a low-level model of computation, whose correct execution can be checked directly via some probabilistic proof system, implemented by the *back-end*.

The front-end of a zero-knowledge proof system implements the procedure which generates this low-level IR (and corresponding variable assignments) from high-level or application-specific representations. Often, the front-end takes as input a computer program Ψ written in a high-level language such as C, and automatically outputs a description of an R1CS system or instance of arithmetic circuit C whose satisfiability is equivalent to Ψ (e.g., in the sense that for any x, there exists a w such that $\Psi(x, w)$ outputs 1 if and only if there exists a w' such that $C(x, w') = 1$, with an efficiently computably mapping between w and w'). The back-end is then run on the IR generated by the front-end, enabling the prover to establish that knowledge of a w' such that $C(x, w') = 1$.

In some settings (e.g., where only one functionality Ψ is of interest, and Ψ will be executed on different inputs in perpetuity) it may make sense to allow significant human intervention in the front-end, i.e., to hand-optimize the intermediate representation of Ψ.

End-to-end efficiency and scalability is highly dependent on having both efficient front ends and back ends. To achieve optimal performance, it may also be essential to tailor the front-end to the back-end. To give a simple illustrative example, some back-ends may handle addition operations much more efficiently than multiplication operations. If using such a back-end, it may make sense to design a front-end that aggressively minimizes multiplication operations, possibly at the expense of a significant increase in the number of addition operations.

Relevance to Benchmarking. To be useful, any benchmarking framework should allow researchers to demonstrate advances in both front ends and back ends, and should allow researchers to tailor front-ends to back-ends to achieve optimal end-to-end performance. At the same time, there can be value to knowing, for specific important IRs (such as some specific R1CS implementing a range proof functionality), what is the most efficient back-end capable of checking that IR.

Exceptions. Notable, some interactive proofs systems do *not* make use of intermediate representations like circuits or R1CS systems. Instead, these proof systems directly express the output of a computation in a form amenable to efficient probabilistic checking, *without* representing the computation via a constraint system or circuit (e.g., the may directly invoke the sum-check protocol [LFKN92] to directly compute some function of the input, without ever first representing the function as a circuit or constraint system. See for example [Tha13, Section 8.2.1]). These proof systems are not general-purpose (i.e., they currently only apply to specific, highly structured computations), but when they do apply they are significantly more scalable than systems that do utilize IRs. And their generality may increase with continued research.

Hence, while a benchmarking framework should allow researchers to demonstrate and measure innovation within front ends and back ends, it should not *force* researchers to use intermediate representations, or the “front end plus back end paradigm” more generally. This means, for example, that the framework should not demand that costs be reported on a “per gate or per constraint” basis, as that notion of cost only makes sense in the context of an intermediate representation.

Identifying Functionalities for Use in a Benchmark. There is an unlimited space of statements or functionalities that people may want to prove using zero-knowledge proof systems. It is thus difficult to choose a a small number of “representative” statements on which people can evaluate/compare their techniques.

Some applications involve proofs of very simple statements (e.g., range proofs). Others applications involve more complex statements, of cryptographic nature, computable by circuits on the order of a few million gates (e.g., “I know a secret key satisfying some property with respect to some leaf of a given Merkle tree root”). Emerging applications feature huge statements that are not cryptographic in nature. E.g., DARPA’s SIEVE program mentions interest in statements about software (“I have knowledge of a vulnerability in this code base”), statements about computation (“this output was produced by running a specific program on some input”), and statements about sociotechnical interactions (“this computation is GDPR compliant”). In particular, statements about computation are by definition as diverse as the class of all possible computer programs someone might write.

Accordingly, it is futile for a benchmarking framework to attempt to predict all functionalities that people might care about, or to claim to identify a representative set of functionalities. Rather, we suggest to identify a small number of settings that are already known to be useful, and ideally that are the dominant cost in some existing applications that people care about. This means that optimizing for these benchmarks will serve at least some utility in some applications.

Tradeoffs and Scalability. Section 1.1 above lists numerous qualitative properties of proof systems, any of which may important to some applications. Ideally, the benchmarking framework should let researchers to report all of these costs

Further complicating matters, in many applications, it is not the raw costs on a given statement size that matter, but rather *tradeoffs between costs*, or *scalability*. By this we mean, questions such as ”what is the biggest statement a system can handle, given (say) 30 minutes and 16GB for the prover to run?”. The bottleneck for scalability is often prover memory (as opposed to prover time or verifier time), and the relevant question is not how much memory is needed on any fixed statement size, but rather how the memory usage grows with the statement size. In other applications, where the statements being proven tend to be small, the main concerns are often how big is the proof and how fast are the prover and verifier, for a specific fixed statement size of interest to the application.

To be useful in both of the above scenarios, a benchmarking framework needs to allow researchers to convey and compare both scability, and efficiency on specific statement sizes.

Many proof systems exhibit complicated tradeoffs between costs, and these tradeoffs can introduce additional subtleties. E.g., suppose there are optimizations one can do to speed up a prover, but these optimizations increase memory usage. This means the optimizations are essentially only applicable to small enough statements. Other tradeoffs may be in terms of “features” (like public verifiability, or avoiding pre-processing costs) vs. efficiency. A single system could offer a huge number of tradeoffs: in principle, any subset of features/costs may experience a tradeoff against the complement set of features/costs. Further tradeoffs come from the use of recursive composition to reduce the prover’s memory (often, at the cost of prover’s running time).

Should a benchmark come with guidance/insist that people describe tradeoffs in a careful and consistent manner? It is not even clear what this guidance should be, other than "please demonstrate any tradeoffs of a system in a thorough and honest manner."

Specificity of the Functionalities, and of the Solutions. "Functionalities" could be defined at varying levels of specificity. For example, suppose the functionality is "prove knowledge of a pre-image of a cryptographically-secure hash function". Are the proof system designers allowed to choose the hash function, potentially picking one (or making up a new one) that is particularly amenable to efficient processing by their particular techniques?

If yes, how does one quantify the security of the hash function that the researchers designed or chose? That is a delicate issue. How many bits of security should the hash function have, and how much effort should have been devoted to attacking it?

If not, i.e., if the benchmark insists upon the use of specific hash function like SHA3, then the benchmark becomes narrow, no longer capturing the real-world goal of identifying the most efficient system for proving knowledge of a pre-image of some cryptographically-secure hash function (in the real-world, people do have the flexibility to choose their hash function to play well with the zero-knowledge proof).

We propose to address this issue by offering each benchmark at multiple levels of specificity. See Section 2 for details.

Even once one decides how specific the benchmark functionalities should be, there is still the question of how much protocol designers should be allowed to tailor their techniques to a particular benchmark. A benchmark is maximally informative if the costs of a system or technique on a particular benchmark functionality are representative of the costs of the same system or technique on other functionalities that practitioners or researchers may care about. How much should protocol designers be allowed to tailor/optimize a front end or back end for the specific benchmark functionality? At one extreme, if they are allowed to tailor arbitrarily (e.g., producing a description of the IR by hand), then really the techniques are no longer general-purpose at all, and it may not be possible to infer from the reported costs any ballpark idea of how efficient the solution would be in even slightly different contexts. Yet it is also not clear how to prevent researchers from tailoring solutions arbitrarily to the benchmark functionality.

Comparable issues in other domains. Similar issues arise in the MPC setting. For example, there is an extensive literature on Private Set Intersection (PSI). The most efficient solutions solve only PSI, but some of the solutions can be tweaked by hand to solve variants of PSI. Other solutions are circuit-based, so to solve a variant of PSI, it is necessary only to tweak the circuit and not the underlying MPC protocol. Even if one thinks of the circuit-based solutions as "general-purpose", there are entire papers that are essentially devoted to optimizing the circuits computing PSI or variants. Also, some circuit-based solutions are easier to use as subroutines in larger MPC protocols, because they don't actually reveal the output, but rather an encryption of the output.

Cryptographic Functionalities and Their Utility as Standalone Primitives. Cryptographic functionalities (e.g., proofs of set membership, polynomial commitment schemes, etc.) are often of interest—either in part or in total—so that they can be used as one component of a more general or complicated proof system. For example, set membership proofs are often used as a primitive in zk-SNARKs for RAM execution (to ensure that the prover is correctly maintaining the memory of the RAM being executed). And polynomial commitment schemes are a central building block of succinct proof systems for circuit or constraint system satisfiability.

When these cryptographic functionalities are used as components of a more general or complicated proof system, they may require various extra properties. For example, to use a polynomial commitment scheme within a succinct proof system for circuit satisfiability, the scheme may need to have an extractability property, beyond the basic binding property required of any commitment scheme [CHM+19]. In many cases, exactly what properties these functionalities require for standalone use may be regarded as an open question.

In addition, when these cryptographic functionalities are used within a more general or complicated proof system, the inputs to the functionalities may possess highly specific structure. For example, when set membership protocols used within zk-SNARKs for RAM execution, the data universe is often a specific finite field that the zk-SNARK works over. While generic set-membership protocols that make no assumptions of structure in the data universe can be used, this may entail significant overheads (e.g., the introduction of constraints or circuit gates that "serialize" finite field elements into their representations as binary strings).

The upshot is that when defining any cryptographic functionality for use in a benchmark, one may have to make difficult decisions about what properties to require of the functionality, or whether or not to allow assumptions about input structure. For example, if the functionality is "give the most efficient possible polynomial commitment scheme for univariate polynomials of a given degree, or for multilinear polynomials"), should the scheme be required to have an extractability property that is sufficient to use it within known zk-SNARKs for circuit or constraint satisfiability? And for set membership queries, should any structure on the data universe be assumed?

While these questions do not have clear answers, it seems reasonable to require cryptographic functionalities to have the properties that render them maximally useful as standalone primitives given current knowledge. Meanwhile, unless the same input structure appears in the vast majority of applications of a given functionality, allowing protocol designers to assume such input structure significantly limits applicability of the solutions. We suggest that if the assumption of such input structure is to be permitted by the functionality, then this be offered at a very low "specificity level" for the functionality (recall that this proposal recommends that each functionality be defined at multiple levels of specificity).

Cost Accounting. When reporting prover or verifier costs such as runtime, it is essential that benchmarkers make absolutely clear what processes are encompassed by the reported measurement. Common sources of ambiguity in this regard include the following.

- When reporting verifier time, many esearch papers do not "count" the time the verifier takes to process the input when reporting verifier time, instead only counting the time to check the proof (assuming the input has already been processed). This is justified in some applications because inputs are committed, and the verifier need not ever process the full input, just the commitment. But at the bare minimum, anyone reporting costs on a benchmark should be clear about this.
- Setup and pre-processing costs (e.g., how expensive is key generation) must be reported.
- Reported benchmarks should be clear on whether the prover's cost includes the expense of *witness reduction*, i.e., the computation of the low-level witness representation (typically

a variable assignment) given to the back-end prover. When the statement represents a computation, this typically consists of executing the computation and noting all intermediate values and (where relevant) random-access lookups.[5] This cost can be high.

Comparing Proof Systems on Common Hardware. A major goal of any benchmarking framework is to facilitate comparisons between different solutions. A totally fair comparison would run all solutions on the same system, as absolute runtimes are meaningless if one solution is run on "faster" hardware than another solution. Unfortunately, a totally fair comparison is an impossible ideal. Even if solutions are run on the same system or hardware, one solution may be better suited to that hardware (whether because solution designers specifically optimized the solution for that hardware, or because the solution just happens to use operations well-suited to the hardware).

Mandating a specific environment (e.g., a specific AWS cluster) in which to test solutions may seem to at least partially address these issues, as all protocol designers would be free to optimize their solutions to that system as much as they can. However, we reject this approach, both because it leaves the community dependent on the availability of a specific environment and it places a heavy administrative burden on the community, and because any specific environment will always favor certain operations over others.

This issue has no perfect solution, but we favor the approach of simply urging solution designers to make portable and easily modifiable implementations of their solutions available, so that other designers can easily run the solution on any system of their choosing. Solution designers should be following this practice anyway if they want people to be able to use their solutions.

While it is outside of the scope of this proposal, there is ample opportunity for future work to develop tools and common interfaces to facilitate portability of implementations, and hence aid comparison of solutions.

To the extent practical, any concurrent activity on the system should be minimized when benchmarks are run.

Parallelization. This issue may be surprisingly fraught. First, there is an issue of establishing best practices for comparing runtimes. There are research papers that run System A in a massively parallel environment, and compare to another system (System B) that is single-threaded, and simply report total runtimes of A and B as if that is meaningful (and there may not be discussion of whether System B could be parallelized if someone were to put in the effort to do so). Second, and related to the "common hardware" discussion above, there is the issue that different systems may be more or less amenable to different types of parallel hardware (GPUs vs. clusters, etc.) Comparing the runtime of System A on an AWS cluster to System B using GPUs may not be meaningful.

Reported benchmarks should include, at the very least, single-threaded CPU performance on a precisely-specified machine, so as to establish a baseline runtime that can be reasonably reproduced and compared. Of course, implementors can also report parallel runtimes, hardware acceleration, etc., but should be clear when comparing these how much effort has been made to parallelize/accelerate each configuration, and any relevant issues alluded to above.

[5]For example, witness reduction for a SHA-256 circuit will compute the SHA-256 digest as well as all intermediate values of every bit-wise operation, and represent these as variables over some field, perhaps along with additional variables associated with the implementation of Boolean gates over a large field.

2 Summary of Proposed Solutions to Challenges

The previous section outlined challenges pertaining to the following issues:

1. Identifying functionalities for use in a benchmark.
2. Allowing solution designers to demonstrate progress on both front ends and back ends (and their combination), while at the same time not artificially tying the hands of protocol designers to insist that they use intermediate representations or work in the "frontend + backend" paradigm.
3. Questions surrounding specificity in both the chosen functionalities and the solutions for those functionalities.
4. Allowing/encouraging researchers to demonstrate complicated tradeoffs between costs, to demonstrate concrete performance on fixed statement sizes, and to demonstrate scalability.
5. Proper cost accounting.
6. Comparing solutions in a manner that is not overly dependent on the operation environment (e.g., hardware).

Here is a summary of our proposed solution to these issues.

For 1), we take a first step of identifying several functionalities that are already known to be useful. In the category of cryptographic functionalities, we include proof of commitments, proof of set membership, proof of zero-knowledge proof verification (i.e., recursive proof composition), and others that we outline in less detail. For non-cryptographic statements, we include range-proofs, which are already well-established as a useful functionality.

We also discuss possible non-cryptographic functionalities to develop as future benchmarks, including polynomial evaluations, and neural network evaluation.

For 2) and 3), this is something that the community previously deliberated upon, and there is a suggestion in the Implementation Track of the community reference [ZKP19, Section 3.5.3]. The suggestion is roughly to *not* prescribe a level of generality in the benchmark functionalities, but rather offer multiple variants of most benchmarks, at different levels of generality/flexibility. This is indeed the approach that we take in this proposal.

To take a running example, consider proofs of set membership. The most general variant of this benchmark may specify an ideal functionality and merely insist that any solution fulfill that functionality. At this level of generality, it would be up to the protocol designer whether to utilize Merkle trees, RSA accumulators, or any other technique for fulfilling the ideal functionality. At a lower level of generality, the benchmark might insist that the solution be via Merkle Trees, but allow the protocol designer to use any hash function that is collision-resistant with 128-bits of classical (non-quantum) security. A yet lower level of generality might insist that the Merkle tree uses SHA-3 as the hash function. The least general variant is to fix, say, a specific R1CS system or arithmetic circuit, and insist that the proof system be a proof of knowledge that the constraint system or circuit is satisfiable.

Some discussion of this lowest level of generality is called for. While fixing an intermediate representation and studying the performance of various proof systems for that representation can have value, it also come with pitfalls. For example, by fixing an R1CS system (and even insisting that the functionality be specified via an explicit description of an R1CS system), one does not capture backends that use other IRs, or that avoid pre-processing for the verifier by exploiting uniformity in the computation. Even backends that do "prefer" to work with explicitly-specified R1CS systems may have idiosyncratic costs (e.g., some backends may better leverage sparsity of the R1CS system than others), and by not giving people the flexibility to choose the R1CS system, one may be disadvantaging some backends over others.

There is clearly utility in this least general type of benchmark, so we do include it in our proposed functionalities (see Section 3) but we wish to be clear about the above caveats regarding its importance.

Regarding specificity of the solutions, we do not see a way to prevent protocol designers from tailoring their solutions arbitrarily to the benchmark functionalities, so we do not propose an attempt to do so. Rather, the benchmark functionalities should be carefully chosen so that even tailored solutions are both useful and informative.

For 4), we simply propose to encourage researchers to accurately and carefully report on relevant tradeoffs between costs. In addition, many natural benchmarks are naturally parameterized by instance size. We suggest forking on that parameter. For example, benchmarkers may be asked report costs of range proofs both for 32 bits and 64 bits. For set-membership proofs, the instance size may be parameterized by two quantities: the size of the data universe, and the size of the set in which membership is being proven. Forking on instance size should allow systems with superior performance on small, practically-relevant instances to demonstrate that, while systems with superior scalability will demonstrate their strength as well.

For 5), we insist the pre-processing costs and input-processing costs be explicitly accounted for, and more generally we demand that benchmarkers be clear about exactly what processes are being measured when a given runtime or space cost is reported.

For 6), we encourage researchers to make portable implementations of their solutions available, and we insist that even parallelizable solutions (at least) report single-threaded runtimes.

3 Proposed Functionalities

As explained in Section 1.3, there is inherently no single representative statement (or small set of statements) that would enable an exhaustive comparison of any two zero-knowledge proof systems. Furthermore, even when we fix a specific statement, it is not possible to generate 100% fair comparisons, let alone apples-to-apples comparisons between those systems. What we propose is to have, for each type of functionality, a set of nested benchmarking levels, where each level would enable comparison of different points in the trade-off spectrum. On one hand, the top benchmarking level, which is the most abstract form of that functionality, would only fix the statement to be proven and the security level, but would allow the user to decide the methods and algorithms used to implement the functionality. On the other hand, the lowest benchmarking level would be a fixed optimized constraint system or arithmetic circuit (i.e., a specific intermediate representation implementing the functionality), where the user would not be able to change any parameter.

Example: Commitments For example, we can think of the functionality *commitment*, to prove that "*for a given value, x, c is a commitment of the value x*". At the top level, the benchmarking framework would give the statement and a level of, say, 128-bit classical (non-quantum) security, and would allow the user to pick any commitment function of their choice, implement it freely and optimize

	Top Level	Bottom Level
Bit Security	Fixed	Fixed
High-level Statement	Fixed	Fixed
Method Used	Free	Fixed
IR (R1CS, Arithmetic circuit, etc.)	Free	Fixed
Front-end Implementation	Free	Fixed

Figure 1: Each functionality has levels of generality, with each level determined by the parameters listed in the rows of the above table. In this table, *Free* means the protocol designer gets to choose the value of the parameter, while *Fixed* means that the protocol designer has no say in the matter. The rows of the table refer to the following parameters: (1) The *Bit Security* defines the level of security for the whole functionality; a given functionality could have different security levels, but then every sub-level in the functionality will also abide by that security. (2) The *High-level Statement* defines the exact functionality being proven on; specifically the input and output space, as well as the high-level relation between the two. (3) The *Method Used* defines the exact algorithm used to implement the relation in the statement; for cryptographic statements, the method includes the instantiation of the primitive used (SHA256, Pedersen, ECDSA, etc.). (4) The *Intermediate Representation (IR)* defines what language is used to write the statement for the proof, as different systems may use different languages (R1CS, arithmetic circuits, algebraic constraints, etc.). (5) *Front-end Implementation* defines the set of optimizations to be done (even after fixing an IR) that allow for implementing a method in different ways.

for the specific trade-offs of the proving system they are benchmarking. For the second level, the commitment function would also be fixed, say using a *Pedersen Commitment*, yet the user would be able to implement an optimized version of the primitive, as well as choose any desired intermediate representation (constraint system or circuit) implementing the commitment scheme, or possibly avoid an intermediate representation entirely. Finally, for the third and bottom level, the benchmark would run a specific file with a fixed implementation of the constraint system or circuit for a specific commitment scheme, IR, optimization and security level.

It is clear that different functionalities may have different number of levels, as new methods, or submethods arise for that functionality. In Table 1 we outline the different parameters that exist, and show the two extremes of the spectrum: the top level and bottom level.

We note that the lowest level is dependent on the specific IR chosen, which means that several files would need to be made available for benchmarking of systems with different IRs. This means that the lowest level should only be used to compare systems that use the same IR. Furthermore, at each of the higher levels, one should nest further the benchmark for different security parameters, where a user may be interested in benchmarking at 128-bit or at 256-bit security. We believe that this method of benchmarking by levels is broad enough to encompass the different ways a system could be optimized, e.g., for number of multiplication gates / addition gates, for circuit depth, for polynomial degree, etc. It enables to both showcase specific benefits in the trade-off space of a given scheme, i.e., optimizing for the back-end (such as taking advantage of uniformity in the constraint system), as well as comparing schemes based on the optimized constraint system, i.e., optimizing for the front-end.

Remark. We note that the functionalities put forth in this proposal do not aim to deal with the efficiency details associated to the specific intermediate languages or schemes (such as bit serialization); we assume that the data is unstructured. See discussion in Section 1.3 on cryptographic functionalities for justification. Furthermore, it is up to the implementer to ensure that the instances of the statements implemented in the benchmark are secure, according to the definitions of security for that functionality. A future version of this proposal could include agreed-upon definitions of the functionalities in an ideal setting, which may significantly ease comparison of the security properties of different proof systems implementing each cryptographic functionality.

In this section we present an initial set of functionalities that we believe are especially useful today. We also explain what are the different levels of generality that we believe should comprise the basic benchmark for each proposed functionality. We divide the functionalities into two categories, cryptographic statements and non-cryptographic statements.

3.1 Cryptographic Statements

3.1.1 Commitments

This functionality is extremely useful in the context of zero-knowledge proofs, as highlighted in the ZKProof Community Reference document [ZKP19] and as presented in the work of LegoSNARK [CFQ19], a commit-and-prove framework for SNARKs. The framework of commit-and-prove is used widely in many privacy preserving applications, as it is required to prove properties about committed data. Unless specified differently, we suggest the security parameter to be fixed for all levels at 128-bit of classical (non-quantum) security.

Definition 3.1 (Proof of Commitment)**.** *Prove that the element c is a secure commitment of the message m; $c = \mathsf{COMM}(m)$. The commitment is* binding *and* hiding *(one or both of these properties may be computational rather than statistical).*

Top Level. For the top level, the statement is fixed, as defined in Definition 3.1, as well as the security parameter.

Intermediate Levels. For this functionality, there can be more that one intermediate level, depending on the specific method used. If we were to choose a Pedersen Commitment, then the first intermediate level would have this method fixed, but would allow the user to chose on which elliptic curve the commitment is computed; in a second intermediate level, the curve parameters would also be fixed. In contrast, if we were to choose SHA256 as a hash function used in the commitment, then there would be one such level as we would only need to fix the function itself (i.e., compared to SHA256, there are more degrees of freedom when implementing Pedersen commitments).

Bottom Level. For this level, we recommend using a set of fixed files that contain an implementation of the functionality on different IRs. For example, in the R1CS language, we could use the circuit implemented originally in the Sapling protocol [HBHW18].

Remark. We note that the functionality of proof of commitment may involve overlap with the following section on set membership. This is because proofs of set-membership can be implemented via techniques that also involve commitment schemes (and proofs about such commitment schemes). We nonetheless feel that it is worth including proofs of commitments as a stand-alone functionality.

3.1.2 Set Membership

The functionality of set membership is used in many applications to commit to a set of elements. In most settings, the element being proven membership must be hidden, which ensures further privacy. There are many methods to compute a set-membership proof in zero knowledge: RSA accumulators [BBF19, CL02, LLX07], Merkle Tree [Mer87], bilinear-pairing accumulators [CKS09, CPZ18, DT08], and more. Unless specified differently, we suggest the security parameter to be fixed for all levels at 128-bit classical (non-quantum) security.

Definition 3.2 (Proof of Set Membership)**.** *Given a set X, prove that the element $x \in \mathbb{D}$ is in the set X. If x is not in X, then one cannot prove that it is in the set.*

Top Level. For the top level, the statement is fixed, as defined in Definition 3.2, as well as the security parameter.

Intermediate Levels. At this level the method should be fixed, which could mean more than one intermediate level. If we were to use an RSA accumulator, then there would be only one intermediate level. However, in the case of a Merkle Tree, we first fix this method while allowing the user to choose the collision-resistant function of their choice, and then in a lower level, we would fix the function (say, SHA256, or Pedersen Hash [HBHW18]).

Bottom Level. For the bottom level, we recommend using a fixed file for every different IR. We encourage users to submit their implementation files.

3.1.3 Recursive Composition

Recursive proof composition is the functionality of verifying a zero-knowledge proof in zero knowledge, which enables the aggregation of proofs, both in parallel and in series [BSCTV17]. A statement could be one verification, or several, which allows for batching of proofs. In general, applications use recursiveness for either compressing information and verification, such as in Coda Protocol, or for hiding the statement being proven [BCG+18].

In fact, recursive composition can be seen both as a stand-alone functionality, used to aggregate proofs in a generic way, or as a technique to handle other functionalities, such as to prove the two commitments commit to the same thing. We have decided to add it here as a standalone functionality, as it can be a useful primitive for batching of proofs for general-purpose ZK proving systems.

Definition 3.3 (Proof of ZKP Verification)**.** *For some statement P, the payload statement or compliance predicate, and zero-knowledge proof π_P of the validity of $P(x)$, prove, in zero knowledge, that π_P verifies correctly. We write that $\pi = zkPoK = \{(\pi_P, vk_P), () : \mathsf{Verify}(\pi_P, vk_P) = 1\}$. The security property says that if the proof π_P does not verify, then the proof π will not verify either.*

Top Level. At this level, we recommend that a one-level recursion of a single proof is implemented as the statement, at a 128-bit classical (non-quantum) security level.

Intermediate Levels. The interesting aspect of this functionality is the methods that it entails fixing:

- First, the underlying proving system, where the first proof is generated. Ideally one would like to use a succinct scheme, in order to be able to prove any statement without affecting the proof size and verification time, and hence without affecting the proving time of the second proof.
- Second, we need to fix the statement for which the first proof is generated. This is especially meaningful when using a system that does not have a succinct proof, given that a short statement may be used, and hence a very short proof generated, which would cause for the verification statement to run faster. We recommend to fix the underlying statement in order to reach a more "apples-to-apples" comparison, potentially by using one of the other existing functionalities.
- Third, the underlying mathematical object of the proving system. Each proving system comes with an underlying object that defines the efficiency and security. For example, pairing-based schemes must have a choice of pairing-friendly elliptic curves, which affects the specific algorithm of the proof verification statement. Pairing-based schemes use pairings to verify the proof, IOP-based schemes use other algebraic techniques to verify low-degreeness of a polynomial.

Remark. For the second layer proving system (the scheme used for benchmarking, where the proof verification statement is actually implemented), one thing to keep in mind is that different proving systems would be better suited to verify proofs computed by schemes with different underlying objects. For example, if the first proof is generated using a bilinear-pairing scheme, then the second proving system should be one that is optimal for computing pairings inside the statement. This means that the benchmarking framework must enable the user to make a choice suitable for their scheme.

Bottom Level. In this level, all of the previous methods must be fixed, as well as the specific IR, yielding a specific implementation of the recursive composition.

3.1.4 Digital Signatures.

For digital signatures, there is a breadth of potential choices, some of which have completely different functionalities (blind signatures [Cha83], ring signatures [RST01], aggregatable signatures, etc.) As such, ideally, one would like to distinguish them as different functionalities, so that each would have its own chain of levels (albeit a similar one).

In this section, we define only the simplest of functionalities, which is to prove that a given message was signed correctly.

Definition 3.4 (Proof of Signature Verification)**.** *Prove that given a message m, and a verification key vk, a signature σ on m verifies correctly using vk. The security property establishes that a false signature will not pass the proof verification. We write that $zkPoK = \{(\sigma, m), (vk) : Verify_{vk}(\sigma, m) = 1\}$.*

3.1.5 Further Primitives

There are plenty of cryptographic primitives that one would want to use for proving some property in zero knowledge. Here are further functionalities to explore in future versions of this framework:

- Message Authentication Codes, where a proof of correct "tag" is given together with the signature verification proof.
- Diffie-Hellman Key Exchange, where a proof of correct key derivation is given to a third party, as to hide the key but prove a handshake took place.

3.2 Non-cryptographic Statements

Non-cryptographic statements span an infinite spectrum of possible statements, from proving the range of an element, to proving the safety of a software program. Here we briefly present a few interesting functionalities and we encourage the community to give feedback on which other statements may be relevant.

3.2.1 Range Proofs.

Range proofs are probably the most useful and most used non-cryptographic statement in applications today. They are used to prove that some integer (or field element) sits between two other integers (or field elements).

Definition 3.5 (Proof of Integer Range)**.** *Prove that given a range, $[a, b]$, a hidden element x lives within the range.*

Top Level. In the top level, the statement is fixed with a given range.

Intermediate Levels. At this level, the method used for implementing a range proof might be fixed. For example, one could use low-bit representation of the integers (as long as the range can be written with a short amount of bits), and then check for an overflow inside the proving system. There may be a variety of ways to implement such a range proof.

Bottom Level. Again, in this level we aim to enable comparisons of schemes with the same IR, as we would have to fix a specific constraint system for the functionality.

3.2.2 Further Functionalities

As mentioned before, there are infinitely many functionalities that one could test. Here are some that may be relevant for applications of zero-knowledge proofs now or in the future.

- Polynomial Commitments/Evaluation / Interpolation, where a party may want to prove that for a given polynomial, a set of (possibly secret) inputs evaluate to a specific set of values Different applications may require varying degrees, domain and range sizes, for which it can be difficult to generalize for a given functionality.

- Neural Networks, where a party delegating the computation of training on data may want to receive a proof that the training was done correctly, or that the error is as given. In this case, there can be many intermediate levels of generality, depending on the activation function, the structure of the network, and more.
- Software Vulnerabilities / Safety, where a user may want to prove in zero knowledge that a given program does or does not have any vulnerability. This is one application at the core of the DARPA SIEVE program[6], which aims to drastically improve the scalability of proving systems to enable these kinds of functionalities.

End-to-End Applications. We have presented several applications of zero-knowledge proofs as possible functionalities to benchmark proving systems. It could well be, however, that these are not good enough to test a specific system on a real-life application. As such, we recommend that the community submit end-to-end applications to potentially become stand-alone functionalities.

4 Benchmarking Tool

We will provide tools to assist in consistently measuring, analyzing and reporting benchmarks. These tools aim to implement all of the above recommendations and considerations, and to make it easy to apply them to the listed functionalities. Specifically, we will provide two tools:

A *driver tool* which executes the algorithms of the *system under test (SUT)*, and provides them with the requisite inputs (e.g., R1CS files). It outputs *measurements files* containing the raw data from the executions (e.g., timing).

An *analysis tool*, which consumes measurement files, applies statistical analysis, and outputs machine-readable and human-readable summaries.

We make this explicit separation (unlike some existing benchmarking frameworks) for two reasons. First, it enables analysis to be re-done (e.g., with different statistics methods) without repeating the measurements. Second, some SUTs may be difficult to invoke from a driver tool, and are easier to run using some existing system (e.g., a MapReduce job) that produces the measurement files by its own means.

The tooling should support measurement and analysis of timing, of memory consumption (if this can be meaningfully captured the SUT), as well as as be extensible to other metrics (e.g., arithmetic operation counts).

4.1 Measurement file format

We will define a file format to store measurements and details of the system, the machine, and the tests that produced them. Each file will contain the following information:

- Identification of the system under test (name, version, homepage), with a commit hash included if available.
- Properties of the system as defined by the standard (e.g., pre-processing and batching modes).

[6] https://www.darpa.mil/program/securing-information-for-encrypted-verification-and-evaluation

- Identification of the machine running the tests (name, CPU, memory, environment, disk used, etc).
- Configuration options for the system under test, such as compilation flags or number of threads.
- One or several functions and parameters as defined by the standard (e.g., input size).
- A series of measurements for each function and parameters (timing, memory, etc).

The file format will be human-readable, and creating new measurements should be as simple as adding new files to a repository containing all measurements thus far. Then, measurement data files can be inspected, corrected, moved, and stored. We offer to curate and host measurements from the community in a central repository, presumably on Github.

We propose the format of the benchmarking framework Airspeed Velocity as a starting point. It supports recording raw measurements and various metadata. This project provides tools to run benchmarks, store measurements, compute statistics, and report the results, including in the form of HTML pages. It is written in Python making it easy to customize, and the format is simple, allowing us to collect or analyse measurements using custom tools as necessary.

4.2 Driver tool

The driver tool executes the functions of the system under test and collects measurements. It is configured by writing two commands: setup and execute. The setup command may run any non-measured startup or preprocessing procedure, and the execute command must perform the task as specified by the standard. The tool will run the provided commands repeatedly, measure wall clock timing, estimate the maximum memory usage, and record these in a measurement file. The tool will also collect and manage the metadata of the tests and environment.

This procedure introduces a noisy overhead as the driver runs the system through a command. This may be mitigated by pre-starting a process and loading any resources of the system in the setup phase, then wait for a trigger over some inter-process communication. The overhead of the execute phase will be limited to the sending of the trigger.

Alternatively, the tool will also accept a time measurement reported by the system itself. The system should be minimally modified to measure the time taken by its core function, excluding any irrelevant startup and shutdown time. The system must print a special message containing the measurements to its standard output or error (stdout/stderr) stream. The driver will recognize this message (ignoring any other output) and use it in place of its wall clock measurement.

Finally, a system may be modified to output a measurement file in the correct format on its own, or the file may even be crafted by hand without using the driver tool at all.

4.3 R1CS driver and zkInterface

For bottom-level functions defined by R1CS or arithmetic circuits, the driver will also provide a description of test circuits and witnesses. In this case, the system under test remains generic and may support any circuit; it only happens to be tested on standard circuits during benchmarking. The ZKProof.org standard proposal zkInterface can be used to represent the standard circuits and witnesses. A draft of this approach was published as zkinterface-http.

4.4 Analysis tool

The analysis tool will read all available data files. For each series of measurements, it will compute basic statistics and assess the reliability of the series. It will then produce a report including tables and charts. It will export the results in a spreadsheet format (CSV) to allow additional analysis and charting.

As an illustration, the following is a table that the analysis may output to summarize a few statistics, per function+param and per system under test:

Function (Param)	System	Prove (ms)	Verify (ms)
Range (64 bits)	R1CS Groth16 libsnark	1234 ±12	2 ±1
Range (128 bits)	R1CS Groth16 libsnark	1234 ±12	2 ±1
Range (64 bits)	Bulletproofs Dalek	1234 ±12	2 ±1
Range (128 bits)	Bulletproofs Dalek	1234 ±12	2 ±1

where ± denotes standard deviation. The following example table presents more metrics per task, such as memory usage or the count of group operations:

Function (Param)	System	Task	Time (ms)	Memory (MB)	Group Ops
Range (64 bits)	R1CS Groth16 libsnark	Prove	1234 ±12	123 ±12	34k
		Verify	2 ±1	123 ±12	12
Range (64 bits)	Bulletproofs Dalek	Prove	1234 ±12	123 ±12	34k
		Verify	2 ±1	123 ±12	12
Range (128 bits)	R1CS Groth16 libsnark	Prove	1234 ±12	123 ±12	34k
		Verify	2 ±1	123 ±12	12
Range (128 bits)	Bulletproofs Dalek	Prove	1234 ±12	123 ±12	34k
		Verify	2 ±1	123 ±12	12

5 Conclusion and Future Directions

We believe that our initial proposal has the potential to develop into a standard used by the community, or at least to augment the brief discussion of challenges and best practices for benchmarking included in the current community reference [ZKP19]. We recognize that there is much more work to do to bring this proposal to such a state. Indeed, this proposal would benefit greatly from discussions with a diverse array of researchers and practitioners in the ZKProof community.

Much future work remains. Among these, the tooling implementation, as described in Section 4, is work in progress.

Other future directions include settling upon simple best practice recommendations for benchmarking. Examples include: (1) settling on a firm recommendation for all aspects of the environment / hardware that should be reported along with any experimental results (e.g., number of cores, amount of DRAM, whether HyperThreading/TurboBoost is enabled, etc.). (2) Settling on recommended procedures for determining reported runtimes (e.g., run the proof system at least 100 times and report the median). (3) As scalability to larger statement sizes becomes increasingly important, it may be reasonable to insist that all experimental reports explain why bigger statement sizes were not experimented upon. i.e., was the bottleneck space usage, run time, hardware costs, etc.

In addition, as future work we believe there may be utility in having ZKProof host a repository of measurements that will aid the community in making comparisons of proving systems. We also encourage the community to contribute to the set of available functionalities, both by proposing new functionalities, and also by implementing the existing ones in different IRs.

In general, it would be beneficial for this proposal to discuss all of the points outlined in Section 2, and furthermore to have an extended discussion about possible best practices raised in this section. These are not first-time discussions in the cryptography community, and we believe rapid progress should be achievable.

References

[BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*, pages 561–586. Springer, 2019.

[BCG+18] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 820–837, 2018.

[BSCTV17] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica*, 79(4):1102–1160, 2017.

[CFQ19] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. Cryptology ePrint Archive, Report 2019/142, 2019. https://eprint.iacr.org/2019/142.

[Cha83] David Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer, 1983.

[CHM+19] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. Technical report, Cryptology ePrint Archive, Report 2019/1047, 2019, https://eprint.iacr.org/2019/1047.pdf, 2019. To appear in EUROCRYPT 2020.

[CKS09] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In *International workshop on public key cryptography*, pages 481–500. Springer, 2009.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Annual International Cryptology Conference*, pages 61–76. Springer, 2002.

[com19] Zkproof community reference, version 0.2, 2019. Available at: https://docs.zkproof.org/assets/docs/reference-v0.2.pdf.

[CPZ18] Alexander Chepurnoy, Charalampos Papamanthou, and Yupeng Zhang. Edrax: A cryptocurrency with stateless transaction validation. *IACR Cryptology ePrint Archive*, 2018:968, 2018.

[DT08] Ivan Damgård and Nikos Triandopoulos. Supporting non-membership proofs with bilinear-map accumulators. *IACR Cryptology ePrint Archive*, 2008:538, 2008.

[HBHW18] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. Technical report, 2018. https://github.com/zcash/zips/blob/master/protocol/.

[LFKN92] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM (JACM)*, 39(4):859–868, 1992.

[LLX07] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In *International Conference on Applied Cryptography and Network Security*, pages 253–269. Springer, 2007.

[Mer87] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[RST01] Ronald L Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 552–565. Springer, 2001.

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *Annual Cryptology Conference*, pages 71–89. Springer, 2013.

[ZKP19] ZKProof. ZKProof community reference, version 0.2, December 2019. Available at: https://docs.zkproof.org/assets/docs/reference-v0.2.pdf. Updated versions at https://zkproof.org.