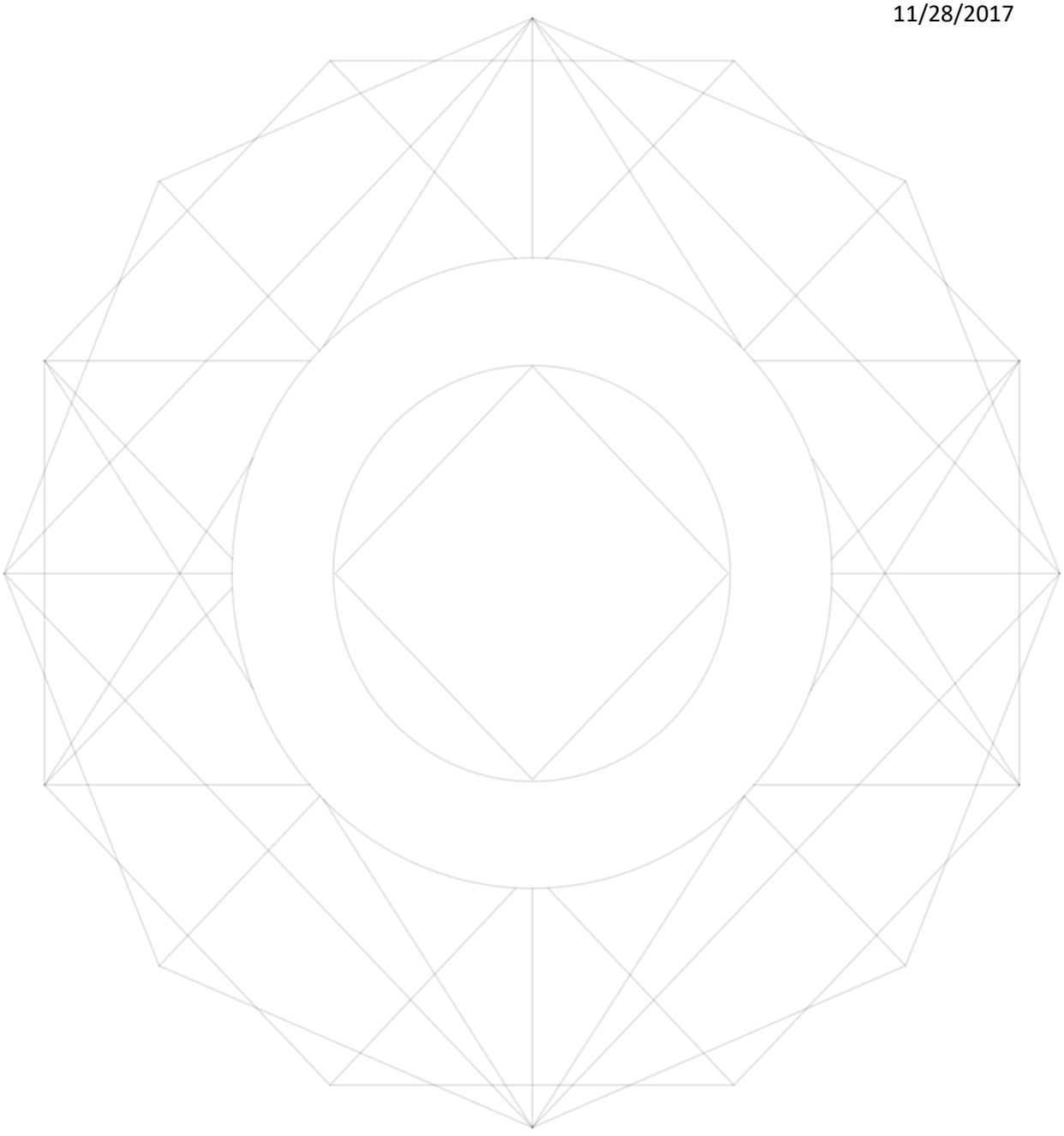
CFX
ALTERNATIVE TRADING

OPEN FINANCE NETWORK

INDUSTRIAL ZERO-KNOWLEDGE PROVING SYSTEM V2.2

11/28/2017



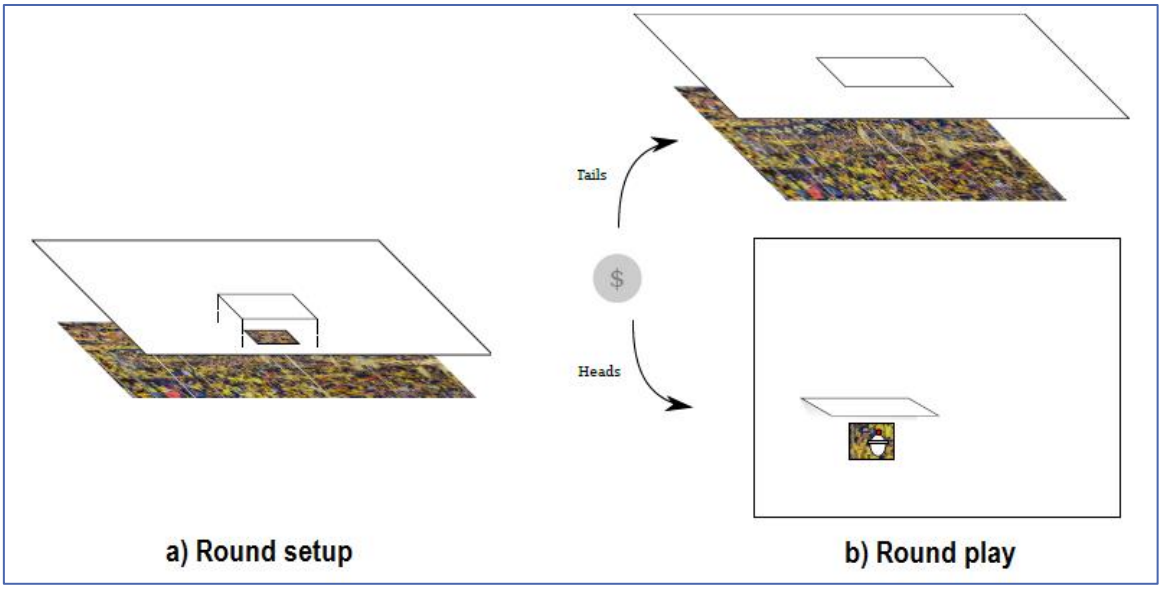
INTRODUCTION

Consider what we might mean by the term “industrial proving system”. The broad functionality is clear. An organization, motivated by the competing aims of transparency and privacy, wants to publish verifiable statements about sensitive data without leaking the data. While it may seem strange at first, it is actually possible to achieve this goal. In this appendix item, we will briefly introduce the concept of a zero-knowledge protocol. Such protocols can be used to build proving systems. We then explore how a proving system might look more concretely, and propose a sensible roadmap for building the Open Finance Network zero-knowledge proving system.

ZERO-KNOWLEDGE ARGUMENTS OF KNOWLEDGE

Zero-knowledge (ZK) protocols are useful when one party, Alfred, wishes to convince another, Brenda, about the truth of a statement without revealing the evidence. ZK protocols were first examined in [GMR89] and a universal protocol was first proposed in [GMW91]. Suppose that there is a puzzle to which Alfred claims he knows a solution. He may want to convince Brenda he knows a solution, perhaps to sell it, without revealing what the solution is. It turns out that there are many things that can be viewed as a puzzle for the purposes of a ZK protocol.

- **Cryptographic digital signatures.** In a digital signature scheme, identity is tied to a public key. The public key is derived from a secret key that only the signer knows. The signer wishes to prove that the signer knows the private key without revealing it. The signatures used by schemes such as GPG, SMIME, and cryptocurrencies can be viewed as ZK proofs of secret key ownership.
- **Anonymous transactions.** One way to achieve financial privacy is to set up a payment system with a “pot” that anyone can pay into. To make an anonymous payment of amount *x* from the pot, a user needs to prove that they have paid at least *x* into the pot (less previous withdrawals). However, to remain anonymous they cannot reveal which incoming payment was theirs. Zero knowledge protocols can unlink payments into and out of the pot. An elaboration on this idea is used by the cryptocurrency ZCash.
- **Verifiable database audits.** A company can use ZK protocols to improve transparency. An audit can be thought of as a program that runs on a database, possibly with some other inputs, and returns whether or not the database is valid, consistent with previous versions, etc. Using a ZK protocol a company can prove that such an audit truly produced a reported result without revealing what the data was.



Let us walk through a detailed example of a ZK protocol. In this example, Alfred and Brenda are playing the game "*Where's Malbo?*". Alfred claims to have located Malbo. How can Alfred prove this to Brenda without giving away Malbo's location? To make sense of this, should have an idea of what it means to give away Malbo's location or equivalently: for one to not know Malbo's location. The standard way to think about this is point out that regardless of what else she knows, Brenda can always randomly guess a location. She has a small chance of guessing correctly, so even in a state of complete ignorance, Brenda might get lucky and find Malbo by guessing. Therefore, when we say that the protocol does not give away Malbo's location, we mean that after the protocol runs, Brenda's best strategy (short of solving the puzzle herself) remains random guessing.

The protocol that follows is an interactive protocol. It requires Alfred and Brenda to perform some, possibly large, number of rounds. To implement the protocol, Alfred and Brenda need a large stack of sheets of paper large enough to cover the Where's Malbo book they are using. They also need a fair coin.

A single round:

1. Alfred takes a sheet of paper and cuts out a Malbo-shaped hole from it.
2. Alfred places the paper with the Malbo-shaped hole over the book in such a way as to reveal Malbo, but cover the rest of the book.
3. Alfred tapes a small flap over the Malbo-shaped hole.
4. Brenda flips the coin:
    - **Heads.** Alfred lifts the flap, revealing Malbo through the hole.
    - **Tails.** Alfred lifts the large sheet, revealing the puzzle underneath.

For an illustration of the setup see figure (1a) and play see (1b). Let us consider the idealized situation where the book is perfectly flat so that Brenda cannot see its outline under the paper. To understand this protocol, we should think about the odds that Alfred can successfully complete many rounds of the protocol without knowing where Malbo is. Suppose that Alfred has complete ignorance about the location of Malbo. On a given round, either Alfred uses the true puzzle or not. If Alfred uses the true puzzle, there is a 50% chance that Brenda will ask Alfred to lift the flap. Given that Alfred does not know Malbo's location the chance that the hole shows Malbo (and not some random swath of the puzzle) is negligible.

In this case Brenda will discover that Alfred cannot find Malbo. Alfred could replace the true puzzle with a forgery in which he knows the location of Malbo. Then there is a 50% than Brenda will have Alfred lift the covering paper and discover the fake. In order for Alfred to go many rounds in this protocol, he must get very lucky: Brenda has to flip heads *every time* Alfred uses the true puzzle and tails *every time* he uses a fake. Since Alfred cannot know in advance the result of the coin flip, the odds of success are 2^{-N} where N is the number of rounds. (With 20 rounds, the odds of successfully lying are worse than 1 in a million.) So, we see that by tuning the number of rounds, Brenda can be sure Alfred is being truthful with as a high a probability as she likes. (If Alfred has some rough idea where Malbo is, his odds of completing the protocol do improve for any given number of rounds.)

At this point, Brenda can be satisfied with the protocol. However, Alfred may have some reservations. To convince Alfred, we must argue that after N rounds of the protocol, Brenda still cannot find Malbo more effectively than random guessing. The key observation here is that the only thing Brenda can see is the sequence of outcomes. While unlikely, it is *possible* that Alfred was totally ignorant, but lucky. This means that if Brenda could reliably extract useful information about the location of Malbo purely from a successful transcript, then she would be able to do so *even in the rare cases where Alfred is totally ignorant*. Thus, the protocol does not leak information.

One drawback of a protocol such as this is that it requires Alfred and Brenda to interact over many rounds. It is possible to convert a broad class of ZK protocols into non-interactive ZK protocols. We can even do this (in an elaborate way) in our charming real-world example. First, Alfred and Brenda need to settle on a random beacon. Suppose that there is a trustworthy service that publishes the outcome of a fair coin toss every second over the radio. Alfred and Brenda can both refer to these coin tosses by the time they occurred. Next, Alfred will need N copies of the puzzle, scissors, tape, and patience. Alfred sends two messages to Brenda.

1. Alfred cuts each of the N puzzles into tiny squares and reassembles them randomly, keeping track of the mapping from original square locations to final square locations. Alfred sends Brenda the stack of scrambled puzzles.
2. Starting at an agreed upon time, Alfred and Brenda note the next *N* coin tosses from the random beacon. For each outcome:
    - **Heads.** Alfred sends Brenda the location of Malbo in the corresponding scrambled puzzle.

- **Tails.** Alfred sends Brenda the instructions to unscramble the corresponding scrambled puzzle.

Both Alfred and Brenda can be satisfied with this protocol for reasons similar to those given above, although at this point they are likely itching to digitize their system! While general purpose ZK proving systems are much more complex, almost all of the central ideas are represented in this sketch. Modern systems have one additional ingredient. The messages that Alfred has to send above are enormous. It is possible to use clever algebra to design *succinct* non-interactive ZK proving systems which have reasonably small messages. Unfortunately, the ideas used to achieve succinctness are outside the scope of this summary.

PROVING SYSTEM ARCHITECTURE

In this section, we discuss some of the properties that a proving system would have to have to be used in production. The term private unambiguously refers to information that should only be known to (some) members of the organization. Public can have a broader range of meanings. For example, in some contexts public information is information that can be known to certain specific parties outside the organization rather than the general Public.

SYSTEM DATA

Computations. A reasonable candidate for the fundamental object in the proving system is the computation, which expresses a function, input types, and an output type. The output type has public designation, but some input types can be private. We expect that computations will be expressed in a special-purpose language.

Proofs. A proof is a data structure that corresponds to a computation, a particular instance of the public inputs, and the output. If valid, a proof implies that there is some value of the private inputs so that the computation transforms the input set to the given output.

Provers. A prover is an algorithm, corresponding to a computation that transforms input values and their output values under the computation into a proof. The prover has access to both public and private data.

Verifiers. A verifier is an algorithm, corresponding to a computation, which validates proofs. The verifier only has access to public data.

While provers and verifiers are algorithms, they are derived from computations. We expect that computations will typically have boolean output, where we would regard them as logical propositions. The central task from the organization point of view should be to declare the

computations and logic for conditions under which to generate proofs. All other elements should be derived and distributed automatically.

COMPONENTS

Proof engine. The proof engine provides two services. It compiles computations into a pair consisting of a prover and a verifier. After compilation the verifier can be distributed to end users. However, construction of proofs, using the prover will probably be an online process. The second service provided by the proof engine is reactive construction of proofs using a stable of provers.

Verification engine. The verification engine will likely have several software representations. In an industrial setting, the verification engine reactively verifies proofs coming over the wire from other organizations. For end users, it is reasonable to compile individual verifiers into e.g. JavaScript for in-browser proof verification.

FEATURES

Automation. Construction and distribution of proofs should integrate seamlessly into the existing technology stack. In an event driven setup, other components can safely ignore the proof engine unless they require a verification or a proof.

Distribution. The distribution model should include specification for provers and verifiers in addition to proofs. All should be treated as versioned data in an ecosystem where several versions (or even standards) may exist.

Modularity. Computation reuse has not been a priority in the academic world. However, it is important to maintain libraries of computations, and provide a means for aggregating computations as much as possible. A growing library of computations and their compiled provers and verifiers that can easily be arranged into pipelines will significantly encourage adoption.

Development. Development in a system with proving has two prongs. First, developers must specify computations. The options for proof specification at the time of this writing are by using a subset of C or by directly expressing the computation in one of several arcane algebraic forms. Neither method has the kind of safeguards one would like to see in a secure computing development environment. Second, developers manipulate proofs by including proof generation and verification into the business logic. In the long term, this would likely be achieved by adding markup to source files indicating whether a proof should be published for a certain function or whether some kind of proof should be requested for a value coming over the wire from an outside system. Then as part of the build/deployment process the markup would be parsed and the proofs automatically made available or requested and verified.

Scalability. The system should make it possible for an organization to provide selective transparency at an extremely granular level. We imagine systems exposing tens, if not hundreds of verifiable, partially blinded computations on their internal data, updated as frequently as technology allows.

SYSTEM ROADMAP

There are already a handful of libraries which implement "research quality" proof systems. The best known of these are libsnark (SCIPR Lab based on [BSCG+13]) and pinocchio (Microsoft Research, [PHGR]). It is an open question whether or not a production ready proving system could be built. However, it is clear that limited versions of the system are possible.

PHASE I: INFRASTRUCTURE

The operating assumption in (I) is that the specification of computations and production of provers and verifiers is an expert task. So it is only possible in this phase to produce a small number of computations using a package like libsnark or pinocchio by ad hoc means. In this situation, the focus is on the system for integrating prover and verifier binaries into the stack. For the most part, development in this phase does not require more than one developer with the sophistication to use and patch the academic libraries.

Construct prover and verifier. Settle on one important and tractable proving task. Devote at least one qualified developer to configuring one of the academic libraries to generate a prover and verifier.

Provisional proof engine. The first implementation of the proof engine can be a wrapper around binaries generated in the previous stage. To the greatest possible extent, the proof engine should be constructed by combining existing task schedulers and event clients.

Provisional verification engine. As in the case of the proof engine the verification engine can also be a wrapper around verification binaries. There must be an emphasis on deterministic program generation. Users must be able to obtain everything they need to build the verification engine from the source code and computation specifications.

Proof exchange protocol. Ultimately zero-knowledge computation is about interactions between several parties. The team should specify a minimal extensible protocol for specifying proofs. This protocol can be used in communications between the proving and verifying engines at different organizations.

Deployment/build tool. Build a tool which analyzes sources in a target language for markup that associates certain computations to certain functions, marks values as private, etc. The tool should generate configuration for the proving engine such that appropriate proofs are generated.

Verification library. A common element of proof-aware programs will be blocking calls to the verification engine which continue upon receipt of some valid proof. So an API must be built which exposes the verification engine to the target language.

PHASE II: DSL AND OPTIMIZATION

The objective of (II) is to provide tooling to *correctly* specify computations and compile provers and verifiers at scale. This probably includes creating a domain-specific language to declaratively represent computations. These representations can then be compiled either to C, then to a prover and verifier using pinocchio, or directly to a prover and verifier. The domain language should sacrifice expressiveness for correctness. Some time should also be spent looking for a way to bootstrap proofs, transforming an existing set of proofs into a new one. An important example is proof updating to reflect dataset updates.

CONCLUSION

We now have a clear idea of the nature of a zero-knowledge protocol. The ability to selectively hide inputs to a computation makes it possible for an organization to become more transparent without compromising confidential data. With a general purpose zero-knowledge proving system, the Open Finance Network could prove any statement about their transaction database that can be verified by a computer program. Beyond digital signature schemes, no such system currently exists. We have described some of the features that would be desirable in an industrial proving system and offer a vision for how to put one together. We are at the beginning of an exciting time as advances in cryptography and computer science make unprecedented levels of trustless coordination possible. The dream of verifiable self-regulation is certainly on the horizon.

REFERENCES

[BSCG+13] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKS for C: Verifying program executions succinctly and in zero knowledge. 2013.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. Knowledge complexity of interactive proof systems. 1989.

[GMW91] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity. 1991.

[PHGR] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio, nearly practical verifiable computation.