Efficient Proofs of Knowledge

Gilbert Pajela, John Downs

12/14/2015

Abstract

Classical interactive proof systems, while secure, are often not sufficiently efficient for a number of applications. By making some reasonable assumptions about the computational power of an honest prover/verifier system, it is possible to design very efficient protocols. In this paper, we will examine several protocols for efficient interactive proof systems. The key application is a secure and certifiable delegation of computation.

Introduction

Perhaps the most obvious use of an interactive proof system is authentication. The prover wishes to demonstrate that it knows something special tied to its identity. The other, and possibly more interesting use is to securely delegate computation. A well known real world example is the Berkely Open Infrastructure for Network Computing (BOINC). BOINC is perhaps most famous for SETI@home, where it would take advantage extra CPU cycles on client machines. It suffers an unfortunate weakness where it trusts the computation of the clients. Privacy minded extraterrestrials, given sufficient resources, could corrupt the clients and provide computations that obscure rather than reveal their communications.

In circumstances where the adversary might be more malicious, it would help if we could verify the computation. However, efficiency is of the utmost concern here. If it takes as long to verify the computation as it does to do the computation, you might as well have done it yourself. A similar concern exists if the verifier has limited computational resources. If the verifier is too limited to complete the verification step, the protocol is useless against a cheating prover.

This paper examines protocols that make efficiency improvements over classical ones. While these new protocols assume that an honest prover has some limited computational power, they remain secure against cheating adversaries with unbounded resources.

1 Efficient Zero Knowledge Proofs [1]

1.1 Motivation

Interactive proof systems have the following form. A system (P, V) consists of a prover P and a verifier V. The prover will demonstrate it knows something in a way that V can confirm. This usually happens by P committing to some value, V proposing a challenge and P, revealing the commitment based on the challenge.

It could be the case that P is lying about it's claim to know some fact though. Usually given a single interaction in this sort of proof system, P could guess correctly half the time. In order for V to really be convinced, the probability that P gets away with lying needs to be negligible. This means running the protocol some k times in order to achieve a probability 2^{-k}. Supposing the question at hand is circuit satisfiability, where one must decide if there exists an input to a circuit C such that C will output 1, then a naive approach would require $O(nk)$ bit commitments for a circuit with depth n. It turns out we can do better with some clever techniques. Moreover, this applies to all NP languages.

1.2 The Result

In particular, for the circuit satisfiability problem, it is possible to construct a zero knowledge protocol that uses $O(n^{1+\epsilon} + (lgn)^{O(1/\epsilon)}k)$ bit commitments, for some positive constant *epsilon*. Unfortunately, such a proof might have an initial cost that is unreasonably expensive in many cases. However, we can make a weaker claim and provide an even more efficient solution. By making an argument instead of a proof; that is to say by limiting the computational power of the prover, and making use of a family of collision intractable hash functions, it is possible to construct an argument for circuit satisfiability commiting only $O(lg^c n)l$ bits per round, where c is some constant and l is the security parameter.

1.3 Methods

In order to achieve the results, we need to understand notarized envelopes and transparent proofs.

A notarized envelope is a zero knowledge proof on a committed bit. It has the advantage of allowing proofs on a set of those bits that state some predicate holds while remaining zero knowledge.

In more detail, notarized envelopes use pair blobs to make zero knowledge equality assertions about two committed bits. A pair blob is simply a random pair of bits whose xor has the value of the bit committed to. For example, if the prover wished to commit to 0, they would randomly choose either $\{1, 1\}$ or $\{0, 0\}$ and commit each bit in the pair as usual for a commitment scheme. The proven can then reveal those two bits and the verifier can compute the chosen bit.

In order to prove equality of two bits x, y, P computes $(x_0, x_1) = x$ and $(y_0, y_1) = y$, and then sends $z = x_0 \oplus y_0$ to V. V replies with a challenge $b \in \{0, 1\}$. P reveals x_b and y_b. V accepts if and only if $z = x_b \oplus y_b$. Where many iterations of this protocol are required, P makes a random pair blob with the appropriate constraint at each step in the same way it might make a permutation of a graph when proving isomorphism.

It is clear by inspection that this protocol is complete. It is sound given enough iterations, because V will reject if $x \neq y$ with probability $p \geq 1/6$. Finally, it is zero knowledge because by only revealing one bit of the pair blob at each step and using a different, independent blob for each subsequent step, $V's$ view is independent of the values of x, y.

Equality proofs using notarized envelopes can be used as a primitive for proving arbitrary circuits are satisfiable without revealing the input. This protocol is again zero knowledge, and thus has an expected polynomial time simulator. This simulator will come in handy later.

1.4 Notarized Envelope Example

Consider a zero knowledge proof of graph 3-colorability [1]. A graph is n-colorable if for all vertex pairs with an edge between them, the vertices have a different color using n-colors. In this protocol, P commits a permutation χ of the graph G that it knows the 3-coloring for. V selects an edge at random from G and P reveals the corresponding vertices in the permutation. V accepts if the two colors are different.

Note that each iteration of this protocol requires P to create another permutation, lest it be revealed. By using notarized envelopes, where a blob is used to commit to the coloring, that permutation step can be omitted. P instead proves that the two colors are not equal using the equality proof above. The result is that, while the original protocol runs in $O(nkm)$ time for a graph with n nodes, k edges, and 2^{-k} error probability, the modified protocol runs in $O(n + km)$ time.

1.5 Transparent Proof Example

A transparent proof is similar to a witness for a statement in NP. The difference is that they are slightly longer at $O(n^{1+\epsilon})$, $\forall \epsilon, \epsilon > 0$ and a circuit size of n. They are checkable in $O(lg^{O(1/\epsilon)})$ time by randomly sampling bits in the string. One consequence of this formulation is a more expensive initialization step that can be run once, resulting in faster runtimes.

In this case, we wish to demonstrate $x \in L$ for some string x and language L. P begins by encoding x' by applying some polynomial time algorithm to x, commits to some transparent proof w and sends that commitment to V. The initialization step occurs only once, at the beginning of the interaction. V responds with a challenge $r \in \{0, 1\}^{log^c n}$. P responds with a zero knowledge

[1] An interactive demonstration of graph 3-colorability is available at http://web.mit.edu/ ezyang/Public/graph/svg.html

proof that V would have accepted. This is repeated $24k$ times to achieve an error probability of 2^{-k}.

1.6 Communication Efficient Arguments for NP

Consider a zero knowledge argument protocol that begins by P generating a blob for the proof and sending that to V. V replies with a challenge b, and P responds appropriately. It is assumed that it is computationally difficult for P to cheat, making this an argument instead of a proof. There is an inefficiency here, where in each iteration of the protocol, the entire proof string is sent to the verifier, $O(n^{1+\epsilon}l)$ bits. But the rest of the protocol only requires $O(log^c l)$ bits. This can be improved by using a Merkle trees. Essentially, the tree is a binary tree that is computationally difficult to change, because each non-leaf node is hashed with the value of it's children, using a collision intractable hash function. The prover can commit the root of the tree and share it with V. Then at the reveal step, P can reveal just the path of the tree that was chosen.

2 Computationally Sound Proofs [2]

2.1 Definitions

A proof of a statement S is computationally sound if it is in the form of some short string σ that is easy to find, even easier to verify and provides computational assurance of it's truth. More precisely, it must be possible to find σ in time similar to that needed to accept S.

2.2 Requirements

By considering efficiency relative to the difficulty of the problem at hand, we can develop an understanding of computationally sound proofs for all true statements. The first requirement of such a proof system is that verifying must be simpler than accepting in all cases. Additionally, the prover's required power must be similar to that required for accepting. Any proof system that meets these requirements is called Feasibly Complete. Finally, the system must be applicable to all semirecursive languages.

Semirecursive languages can be defined in two ways, with respect to either accepting or verifying. The first definition states a language L is semirecursive iff there exists an accepting Turning machine such that for all $x \in L$, $A(x) = YES$. The other states that L is semirecursive iff there is a verifying Turning machine that halts on all inputs, where for each $x \in L$, there is some binary string σ where $V(x, \sigma) = YES$.

2.3 CS Language and Proof Systems

The first construction we require is a CS language. The language is the set of all quadruples $q = (M, x, y, t)$ with the following constraints. M is some Turing

machine, x and y are binary strings, and t is a binary string representing an integer no longer than either $|x|$ or $|y|$. Finally, $M(x)$ must output y in t steps.

An oracle is a function from binary strings of length a to binary strings of length b. If an algorithm A calls some oracle f, this is written A^f. If A calls multiple oracles f_1, f_2 this is written A^{f_1,f_2}. For simplicity, an oracle query is considered as a single step, although it could be polynomial in a and b without effecting these results in any fundamental way.

$P(.)$ and $V(.)$ are two Turing machines using random oracles, where $V(.)$ runs in polynomial time. (P, V) is a CS proof system with a random oracle if the system has feasible completeness and computational soundness.

(P, V) requires a shared oracle f, a quadruple q that P claims is a member of L, and a security parameter k. $P^f(q, 1^k)$ will execute and produce a CS certificate C that $V^f(q, 1^k, C)$ will verify.

Because we only have computational soundness, we leave room for erroneous proofs, but in practice the probability of finding one is negligible, so they will not be seen in practice. This means that while CS proofs may be inconsistent, they are computationally indistinguishable from consistent systems. This is achieved simply by making cheating hard enough by increasing the security parameter than no actual prover has the resources to cheat effectively.

2.4 A Computationally Sound Protocol

A CS Protocol can be constructed by making modifications to the efficient zero knowledge argument protocol described above. The input is a quadruple q that is a member of the CS language. To generate a proof, P runs $M(x)$ and records the history of t steps as σ. This record is a proof that $M(x) = y$. σ can be converted into a probabilisticly checkable form t in time $poly(t)$. Then t is converted to a Merkle tree, so only the root needs to be shared initially. V responds with a random tape corresponding to the values it wishes to sample, and P reveals the appropriate path on the tree.

By using a random oracle, it is possible to make this protocol non-interactive. P will query the oracle for some k random tapes and produce the correct responses for them. P then sends all k responses to V, who can use the same oracle to verify the proofs.

In order for P to cheat in this scenario, it would need to be able to find a collision in the collision intractable hash functions used to create the Merkle trees. The probability of the malicious prover being able to find such a collision is negligible. Even if it were to succeed in finding one, it would still need to perform this feat k times.

2.5 Certified Computation

Certified computation asks the question "is y the correct output for algorithm A with input x?". This is both useful for verifying that the underlying implementation of A is functioning properly, and not being sabotaged by hardware

failures or cosmic rays [2]. We may also be interested in proving the computation was correct without requiring the verifier to perform it.

Such a system might be constructed by augmenting A as A' during compilation. In addition to computing what A computes, it outputs a certificate C, that can be verified by a CS verifier V. As long as V is developed using standard formal methods for assuring correctness, we can be confident when it asserts some computation certificate is valid. This is because the likelihood of finding another string σ that collides with C is negligible.

The downside to such a system, besides the cost of producing a certificate being about equal to calculating $A(x)$, is that their construction relies on the execution history of a Turing machine. This is certainly not a practical implementation tool. Any practical construction would require an similar record of computation that was as rigorous.

3 Interactive Proofs for Muggles [3]

Traditional interactive proofs assume a computationally unbounded prover, but it is possible to construct these proofs with a polynomial time prover,a Muggle instead of Merlin, and a linear time verifier.

The primary application in this case is proving the correctness of delegated polynomial-time computations. In order to make delegation worthwhile, a method needs to be extremely efficient. Particularly, the verifier should run linearly in the input size and polylog in the size of the computation of L. The prover complexity is polynomial in the size of the input, and the communication complexity is polylog in the size of the computation.

Applications are large scale distributed computing such as Berkley Open Infrastructure for Network Computing (SETI@home) and communication with peripheral devices that may have very constrained computational power.

A key point to consider are that it must be sound against dishonest verifiers who may have unbounded computational power.

For any langage L that can be computed by a family of ($O(log(S(n)))$-space uniform boolean circuits of size $S(n)$ and depth $d(n)$ has an interactive proof with the following properties.

1. The prover runs in time $poly(S(n))$ and the verifier runs in time $n \cdot poly(d(n), logS(n))$ and space $O(log(S(n)))$.

2. The protocol is complete and sound.

3. The protocol is public coin with communication complexity $d(n) \cdot polylog(S(n))$

The goal is to reduce V's runtime to be proportional to the depth of C instead of it's size and to not significantly increase the runtime of P.

[2] Perhaps caused by our introverted alien friends from before, or malicious Emacs butterflies https://xkcd.com/378/

3.1 Non Interactive CS Certificates

3.2 Public Coin Log Space Verifiers

3.3 Non-Uniform Circuit Families

3.4 Short Zero Knowledge Proofs

3.5 IPCP and PCA

3.6 Methods

3.7 The Protocol

References

[1] Joe Kilian, *A note on efficient zero-knowledge proofs and arguments (Extended Abstract)*, 1992.

[2] Silvio Micali. *COMPUTATIONALLY SOUND PROOFS* 2000.

[3] Shafi Goldwasser, Yael Tauman Kalai and Guy N. Rothblum *Delegating computation: interactive proofs for muggles* In Proceedings of the ACM Symposium on the Theory of Computing (STOC), 2008.