Efficient Proofs

Gilbert Pajela, John Downs

12/14/2015

Abstract

Computationally efficient zero knowledge protocols are neat...

1 Introduction

2 Efficient Zero Knowledge Proofs

2.1 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.

2.2 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.

2.3 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.

2.4 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

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

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.

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

2.6 Efficient arguments for NP

We can transform the above zero knowledge proof into a computationally zero knowledge argument.

...

3 Computationally Sound Proofs

3.1 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.

3.2 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. Finally, the system must be applicable to all semirecursive languages.

Semirecursive languages are...

Feasible completeness requirements...

3.3 CS Language and Proof Systems

The first construction we require is a CS language. The language is the set of all quadruples $c = (M, x, y, t)$ 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.

Formally, this is...

(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.

3.4 CS Proofs are Efficient

3.5 P and NP

CS Proofs are important and meaningful even if P = NP!

3.6 Interactive CS Proofs

An interactive CS proof is ..

Example construction...

Random Strings...

3.7 Computationally Sound Checkers

Validating one sided heuristics... CS checker definition CS checker implmeneting SAT

3.8 Certified Computation

Constructing certified computation systems

Pros and cons

Implementations

4 Interactive Proofs for Muggles

- 1. intro - complexity-theoretic setting - cryptographic setting - delegating computation - roadmap for section 1 - 1.1 main result - thm. 1.1 - corollary 1.2 - comparison to prior work on interactive proofs - comparison to prior work in other models - 1.2 one-round arguments - thm. 1.3 - applying kalai and raz (2009) to other interactive proofs - 1.3 public coin log-space verifiers - corollary 1.4 - 1.4 non-uniform circuit families - thm. 1.5 - 1.5 succinct 0kn proofs - thm. 1.6 - 1.7 - 1.6 results on ipcp and pca - low communication and short interactive pcp - pca w/efficient provers - 1.7 subsequent work - 1.8 bird's-eye view of the protocol - the big picture - going from layer to layer - utilizing uniformity - organization of the exposition - 2. preliminaries - 2.1 turing machines, circuits, and complexity classes - 2.2 interactive proofs - def. 2.1 - 2.3 low-degree extension - prop. 2.2 - proof - claim 2.3 - proof - 2.4 low-degree test - lemma 2.4 - 2.5 interactive sum-check protocol - lemma 2.5 - proof - 2.6 private information retrieval (PIR) - def. 2.6 - 3. the bare-bones protocol for delegating computation - thm. 3.1 - 3.1 preliminaries - parameters - assumptions and notations - 3.2 the bare-bones protocol - protocol overview - the i-th phase - the d-th phase - the final verification - 3.3 proof of thm. 3.1 - completeness - soundness - complexity - 4. interactive proofs: implementing the bare-bones protocol - conventions: a recap - 4.1 interactive proofs for nl - 4.1.1 circuits for nl languages w/efficient low-degree add_i, $mult_i$ - overview - step 1 - lemma 4.1 - proof - preliminaries - the circuit c - the constant gates - the input sub-circuit - intermediate sub-circuits - the machine g - step 2 - claim 4.2 - proof - step 3 - lemma 4.3 - proof - 4.1.2 realizing the bare-bones protocol - thm. 4.4 - proof - corollary 4.5 - 4.2 interactive proofs for l-uniform circuits - claim 4.6 - proof - thm. 4.7 (1.1) - proof - corollary 4.8 - thm. 4.9 - 4.3 protocols for delegating non-uniform computation - thm. 4.10 (1.5) - proof - 5. low-communication 0kn interactive proofs - notation - thm. 5.1 (1.6) - thm. 5.2 (1.7) - proof (idea) - proof of thm. 1.6 - proof of thm. 1.7 - 6. one-round arguments for delegating computation - thm. 6.1 [kalai and raz 2009] - thm. 6.2 (1.3) - 7. an interactive pcp - 7.1 preliminaries - def. 7.1 [kalai and raz 2008] - 7.2 new improved interactive pcps - thm. 7.2 - corollary 7.3 - remark - thm. 7.4 - proof outline - comparison w/the scheme of kalai and raz [2008] - proof of thm. 7.4 - parameters - analysis of the protocol - 8. a probabilistically checkable argument - def. 8.1 [kalai and raz 2009] - remark - thm. 8.2 [kalai and raz 2009] - thm. 8.3 - remark