Proving a Theorem in Zero-Knowledge

J.W. Pope

September 2, 2004

Abstract

In 1986, Manuel Blum claimed in his paper "How to Prove a Theorem So No One Else Can Claim It" that he could apply zero-knowledge methods to proving theorems in any proof system. He sketched a proof of this result, but gave no details. We give a background of zero-knowledge proofs and examine their applicability to the propositional calculus. We then examine Blum's claim, assuming the existence of a proof-verifying program which can be modeled with a finite state machine, and complete the proof. However, we also show that a zero-knowledge proof demonstrating knowledge of a proof is no easier than actually determining a proof of a theorem from scratch.

1 Introduction

Ever since the concept of zero-knowledge was first articulated, a large number of problems that can be exploited for their use have been identified, and a large number of applications determined. In particular, it has been determined that the entire class of NP-complete problems have associated zero-knowledge proofs, subject to a few assumptions [7, 8]. One of the most interesting applications, however, is the idea that actual mathematical theorems are provable in zero-knowledge. This idea was first put forth by Blum in [2], but he goes into little detail of just how this is to be accomplished, although as Bruce Schneier points out in [11], Blum could have published these results without revealing them! A search of the literature has turned up no additional writing on the subject save for references to [2]. Schneier states that the protocol involves mapping the problem onto the Hamiltonian cycles problem, whereby successfully proving a theorem becomes equivalent to finding a Hamiltonian cycle in a graph although this is not clearly stated in [2]. In short, while it seems to be accepted that proving theorems in zero-knowledge is possible, those who know how this is done have not seen fit to tell the rest of us.

In order to fill in these gaps, we start by assuming that there exists some program for checking a proof that can be run on a computer with finite memory. This computer program can be modeled as a finite state machine. This has the effect of immediately shedding some light on the problem. A finite state machine effectively maps the problem onto a graph, and transforms it into the

problem of finding a path through this very complex graph to a certain vertex - a good setting for a zero-knowledge proof. In this paper, I demonstrate how a zero-knowledge protocol can be used to show knowledge of a proof without revealing it. However, this brings up an unfortunate complication. Finding a path between two points on a graph is not an intractable problem. Therefore the zero-knowledge proof as it is formulated is not any easier than simply proving the theorem.

2 Zero-Knowledge Proofs: Basic Concepts

Information is an intangible, ephemeral commodity, and virtually impossible to control. Conversations can be overheard, letters intercepted, and e-mail servers hacked. Of course, good cryptography can keep the information a secret between two parties. But what is there to stop this second party from transferring that information to a third party, who can then in turn pass it to a fourth, and so on? In some cases, it is desirable to demonstrate that you possess some information, but not to reveal it. For example, some authentication schemes require you to give your private key to a trusted authority, not because the trusted authority needs to know it, but because it needs to be certain that you do, in order to prevent potential attackers from exploiting the certification process. This is where zero-knowledge comes into play.

2.1 Definition

A zero-knowledge proof is an interactive protocol between two parties, called the "prover" and the "verifier". The prover claims to have some information, and the verifier would like to confirm that she actually knows it. The prover, however, does not want to reveal it. In order to satisfy their respective desires for secrecy and verification, the prover and the verifier engage in an interactive proof system. This proof system should have two properties. It should be "complete", i.e., if the prover really does have the information claimed, the verifier should come out of the protocol convinced of this fact with a probability of 1, and it should be "sound", i.e., if the prover does not have the information claimed, the verifier should be able to see to it that the probability that she can get away with it is arbitrarily small. (We follow here the terminology of [12]; "validity" is sometimes used in place of "soundness", as in [7].)

These criteria, however, do not form the basis of zero-knowledge. The verifier emerges from the protocol convinced that the prover possesses the information, but what is to prevent him from using the session transcript to convince a third party that he possesses the information himself? Suppose, then, that the verifier could provide a falsified transcript to the third party. If it is computationally tractable to do so, then the third party will not be able to trust any transcript the verifier may provide. Therefore, we call an interactive proof system a zero-knowledge protocol if and only if there exists a simulator that can output results identical to those of a real protocol.

An example will serve to illustrate these ideas on an informal level. Suppose that the prover claims to know a Hamiltonian cycle C to a graph G. The prover and the verifier execute the following protocol, which was originally developed by Blum in [2] as an improvement on an earlier protocol appearing in [7]:

1. The prover applies a random permutation ϕ to G. She then encrypts each entry in the adjacency matrix of $\phi(G)$ using a bit-commitment scheme (about which more shortly), and passes the resulting ciphertext $E(\phi(G))$ to the verifier.

2. The verifier determines a random bit i, and transmits it to the prover.

3. If $i = 0$, the prover decrypts every entry in the adjacency matrix and reveals the permutation, enabling the verifier to determine for himself that $\phi(G)$ is isomorphic to G. Otherwise, if $i = 1$, then the prover decrypts the $|G|$ entries in the matrix that correspond to the edges in the Hamiltonian.

4. This procedure is repeated for a total of k rounds.

It all hinges on the fact that the prover hands over the permuted graph before the verifier chooses what he wants to see. Suppose the prover didn't know a Hamiltonian cycle for G. She has two choices: She can give the verifier a permutation of G for which she still does not know a Hamiltonian, or she can give him a graph the same size as G for which she does know a Hamiltonian, but which is not isomorphic. Because she doesn't know what the verifier wants to see (and since the verifier chooses randomly each time, she has no way of guessing in advance), she will choose correctly with a probability of $1/2$ each round. If she chooses wrong, the protocol ends in failure then and there. Thus, this protocol can be seen to be sound: the probability that the prover can convince the verifier that she knows a Hamiltonian she doesn't know is $1/2^k$ after k rounds. The verifier can simply choose $k \geq -\log_2 M$, where M is his desired level of probability. The protocol is trivially complete— there is simply no way for step 3 in the protocol to fail, if the prover really possesses the Hamiltonian cycle. As for the prover, she is protected by the inherent difficulty of the problems involved. In the first case, the verifier will learn a permutation of G, but this is of no help to him in finding a Hamiltonian cycle. In the second case, the verifier will learn a Hamiltonian cycle, but will have no way of matching it to any set of vertices and edges in G. However, it is not this fact which determines zero-knowledge.

The zero-knowledge properties stem from the simulatability of the entire process. In a genuine instance of the protocol, a transcript will be produced documenting the information available to the verifier during each round. It will contain the graph G and the encrypted permutations $\phi(G)$, the random bits i, and the permutations and Hamiltonian cycles that the prover reveals in step 3 in each round. Suppose the verifier wished to forge a transcript. He knows the graph G. He does not know a Hamiltonian cycle for it, but he can easily create a second graph H which is the same size as G and has a Hamiltonian cycle, which

he does know. Due to the NP-completeness of the Graph Isomorphism problem, it will be infeasible for any other computationally limited party to show that G and H are not, in fact, isomorphic.

The verifier now has everything he needs to generate a false transcript. He first simulates the prover's role. He determines a random bit h and a random permutation ϕ. If $h = 0$, he enters $E(\phi(G))$ onto the transcript. If $h = 1$, he enters $E(\phi(H))$ onto the transcript. Now, he simulates his own role. He determines a random bit i. If $h = i$, then he has asked the simulated prover either to demonstrate the isomorphism of $\phi(G)$ and G, or to show the Hamiltonian cycle in $\phi(H)$. So, he simply enters i onto the transcript and concludes the round by having the simulated prover decrypt the relevant information. If $h \neq i$, then he has either just asked to reveal the Hamiltonian in $\phi(G)$, or to demonstrate the isomorphism of $\phi(H)$ and G. He cannot do either of these things, but he can reset the simulator and start the round from scratch.

Given that the probability that he will need to restart a round is $1/2$ each time, the expected running time for this simulator is twice that of the expected running time of the actual protocol. And the final result is a transcript that could have been generated in a real protocol as far as any third party who cannot decrypt the graphs could determine. (The conditions under which a third party can decrypt the graphs are discussed in the section on bit commitments.)

Of course, this only applies if the verifier is really choosing his bits randomly. After all, the protocol guards against cheating provers, but has nothing to say about cheating verifiers. He might choose his bits as a function of previous bit choices and graph permutations. However, this can also be simulated, as we shall see. Thus, since the verifier cannot prove any transcript, real of false, to be genuine, he cannot simply turn around and claim that he knows the Hamiltonian cycle to a third party. Put another way, the verifier learns nothing from a genuine instance of the protocol that he could not learn from a spurious one! It is this idea that is the essence of zero-knowledge, and it will be developed more rigorously in subsequent sections.

2.1.1 Computational Assumptions

What can we say about the computational powers of any prover or verifier in the above protocol? The prover is obviously limited in his computational ability—he cannot solve NP-complete problems, otherwise there would be little point in concealing the Hamiltonian cycle from him in the first place. The prover, on the other hand, only receives one randomly generated bit from the verifier. We can assume that she has unlimited computational power, and it will not affect the protocol. This holds for all members of the class of zero-knowledge proofs. There is a second, related class called "zero-knowledge arguments" in which the opposite is assumed: limited prover, omnipotent verifier. This variant will be discussed more fully below.

2.1.2 Types of Zero-Knowledge

In the informal discussion of the zero-knowledge properties of the Hamiltonian cycle proof above, we mentioned that the transcripts generated by the simulated protocol were identical to the transcripts generated by the actual protocol. However, we would like some additional constraints. Certainly, the transcripts themselves are identical, but what about their probability distributions? In the long run, can we expect to see a certain transcript the same number of times whether it is being generated by a real protocol or a simulator? To address this question, Goldwasser, Micali, and Rackoff listed three categories of zero-knowledge proofs: "perfect", "statistical", and "computational" [9]. A fourth category, "no-use", covers a range of proofs that fall somewhat outside the formal definition of zero-knowledge, but still do not impart any useful information to the verifier [11].

A proof is said to be "perfect zero-knowledge" if and only if there exists a simulator which runs in polynomial time and yields transcripts with an identical probability distribution to the genuine protocol. An example of a perfect zero-knowledge proof is that for the Graph Isomorphism problem from [7], in which the prover wishes to demonstrate knowledge of a permutation π which transforms the graph G_0 to G_1:

1. The prover applies a random permutation ϕ to G_1, yielding H. She transmits H to the verifier.

2. The verifier determines a random bit i, and transmits it to the prover.

3. If $i = 0$, then the prover reveals the composition $\pi \circ \phi$, and the verifier checks that $\pi \circ \phi(G_0) = H$. If $i = 1$, then the prover reveals ϕ, and the verifier checks that $\phi(G_1) = H$.

4. They repeat this procedure for a total of k rounds.

The protocol is trivially complete. Soundness follows from the fact that if the two graphs are not known to be isomorphic, then the prover will have to decide in advance which graph to permute and transmit, and there is a one in two chance each round that the verifier will want to see the permutation for the wrong one. As for zero-knowledge, just as with the Hamiltonian protocol, the verifier can set up a simulator that chooses a random bit h and permutes G_h at random, chooses another random bit i, and if $h = i$, reveals the permutation, and otherwise restarts the round. This simulator has an expected running time twice that of the genuine protocol, so it runs in polynomial time. As for the probability distribution, in the case of the genuine protocol, the transcript will consist of k triples (H, i, τ), where τ is equal to either ϕ or $\pi \circ \phi$, depending on the value of i. The bit i is randomly generated, as is ϕ, and because H depends on ϕ, we can see that every triple occurs with equal probability. Moving on to transcripts generated by the simulator, every i is equally likely. If $i = 0$, the

permutation that transforms G_0 to H is randomly generated, and just as likely as any other permutation. It will also be equal to $\pi \circ \phi$, for some ϕ not known to the verifier. If $i = 1$, then again, the permutation that maps G_1 to H is randomly generated, and just as likely as any other, and equal to some ϕ. Thus, we see that every triple is equally likely to appear in a falsified transcript as well, and so the two probability distributions are equal— so long as the verifier stays honest.

However, if he chooses his bits nonrandomly, there is still a simulator that can generate the necessary transcripts. Since the probability distributions cannot be analysed directly, they must be analysed inductively on the number of rounds. As the basis of our inductive proof, round zero is simply the beginning of the protocol, when the only items on a real or forged transcript will be G_0 and G_1. The probability distributions are obviously identical in this case. Assume that the probability distributions for real and forged transcripts are identical up to round j. In round $j + 1$, the prover will choose a random permutation of G. The graph $\phi_{j+1}(G_1)$ depends on this permutation, so the probability that a given permutation/graph combination will be chosen is $1/|G_1|!$. The cheating verifier will choose $i_j = 0$ with a probability of p, and $i_j = 1$ with a probability of $1 - p$. Thus, the probability that $(\phi_j, H_j, 0)$ gets written on the transcript is $p/|G_1|!$, and the probability that $(\phi_j, H_j, 1)$ appears is $(1-p)/|G_1|!$. Considering a simulator that models a cheating verifier, the simulated prover is still playing fair, thus the probability that $h_l = 0$, is still $1/2$ for each iteration l. Thus, the probability that $h_1 = i_{j,1} = 0$ is $p/2$ and the probability that $(\phi_{j,1}, H_{j,1}, 0)$ is written to the forged transcript in the very first iteration is $p/2|G_1|!$. Similarly, the probability that $(\phi_{j,1}, H_{j,1}, 1)$ is written on the first attempt is $p/2|G_1|!$. The probability that nothing at all is written on the first iteration, and the round restarts is of course $1/2$, and so the probability that $(\phi_{j,2}, H_{j,2}, 0)$ is written on the second iteration is $p/4|G_1|!$. So, we see that the probability that a triple that includes $i_{j,l} = 0$ is written to the transcript on the lth attempt is $p/2^l|G_1|!$, and therefore, the probability that some triple $(\phi, H, 0)$ appears on the transcript for round j is:

$$\frac{p}{|G_1|!} \sum_{l=1}^{\infty} \frac{1}{2^l} = \frac{p}{|G_1|!}.$$

Through analogous reasoning, the probability that $(\phi, H, 0)$ turns up in the jth position on the transcript is $(1 - p)/|G_1|!$. Therefore, there exists a simulator that produces transcripts identical and identically distributed to those of a real protocol, and therefore, the Graph Isomorphism protocol is a perfect zero-knowledge proof.

A slightly weaker version of zero-knowledge also exists. It may be that a simulator may produce in polynomial time transcripts with a probability distribution identical to the real protocol except in a constant number of instances. These protocols are called "statistical zero-knowledge proofs".

Weaker still are the "computational zero-knowledge proofs". Instead of having probability transcripts with probability distributions identical or nearly identical to the real protocol, their simulated transcripts are merely indistinguishable

from real transcripts in polynomial time. What do we mean by "indistinguishable"? Consider the set T of transcripts with length less than or equal to l. Consider also two probability distributions P_0 and P_1 on T. These two probability distributions are computationally indistinguishable if there does not exist any probabilistic algorithm $\mathbf{D}$, called a "distinguisher", which maps each t in T to $\{0, 1\}$ based on the likelihood that t arose from the probability distribution P_0 or P_1, such that $|E_{\mathbf{D}}(P_0) - E_{\mathbf{D}}(P_1)| \geq |l|^{-c}$ for some large constant "security parameter" c, where

$$E_{\mathbf{D}}(P_i) = \sum_{t \in T} p_i(t) p(\mathbf{D}(t) = 1|t).$$

In other words, the difference between the expected distinguisher outputs taken over P_1 and P_2 should be very small.

Finally, we have the "no-use zero-knowledge proofs". These are proofs that are not necessarily simulatable, but can be established to reveal only insignificant amounts of data to the verifier.

2.1.3 Minimum Disclosure

Alternatively, we can remove the zero-knowledge requirement entirely, but retain completeness and soundness. Such a protocol is called a "minimum disclosure proof". This idea was developed in [6]. (Let the researcher beware: "minimum disclosure" was used interchangeably with "zero-knowledge" in the infancy of the field. Even now, we occasionally see "minimum disclosure" used in this sense by researchers who dislike the term "zero-knowledge".)

2.2 Bit Commitment

In the Hamiltonian protocol above, we stated that the adjacency matrix of the permuted graph was to be encrypted before the prover passed it to the verifier, but we went into no detail about how this was to be done. Encryption in this context is done via a "bit-commitment scheme": a probabilistic encryption algorithm used to encrypt one bit at a time. It takes the form of an interactive protocol which resembles a zero-knowledge proof in that it also makes use of the "cut-and-choose" paradigm. The concept of bit commitment actually predates that of zero-knowledge. It was first put forward by Blum in [1] as a means for two parties conversing on a telephone to flip a coin. In that context, the prover would flip a coin and encrypt the resulting bit by binding it up in some intractable problem, such as quadratic residues or discrete logarithms. She would then send the resulting ciphertext, called a "blob" to the verifier, who calls the toss. Finally, she would "open the blob", i.e. reveal how the bit was encrypted so that the verifier could see for himself whether the coin came up heads or tails. Although the idea of using encryption in zero-knowledge proofs can be seen very early on, the concept and the terms "bit commitment" and "blob" were first formalized in [3].

The role of bit-commitment in the Hamiltonian protocol is similar to its role in flipping coins by telephone: the prover encrypts each entry in the adjacency matrix separately, sends all the blobs to the verifier, and opens the relevant blobs depending on what the verifier asks to see.

As an example, consider the following protocol, based on quadratic residues which appears in [4, 3, 12]. The prover generates two secret primes p and q. The product n of these two primes she makes public. She determines a quadratic nonresidue m modulo n, and this too she makes public. She and the verifier then execute the following protocol.

1. When the prover wishes to encrypt a bit i, she chooses a random x and computes $m^i x^2$, and transmits this blob to the verifier.

2. When the verifier needs to know i, the prover simply reveals it along with x, so that the verifier can see for himself that the bit was encrypted properly.

It is important to note that each bit is to be encrypted separately, with a different value of x each time.

2.2.1 Computational Assumptions

Bit commitment schemes have two essential properties: A bit commitment is said to be "concealing" if the verifier cannot extract the bit directly from the blob, and it is said to be "binding" if the prover cannot encrypt both a one and a zero as the same blob [12].

If we assume unlimited computing power for the prover, it is necessary that any bit commitment scheme must be unconditionally binding— it must be mathematically impossible for the prover to open a blob as both a one and a zero. The example above can be seen to be unconditionally binding. Suppose there was some blob that could be opened as both a zero and a one. Then, we would have $mx_1^2 \equiv x_2^2 \pmod{n}$ for some x_1 and x_2, and so $m \equiv (x_1^{-1}x_2)^2 \pmod{n}$. But, since it has already been established that m is a quadratic nonresidue modulo n, this is a contradiction. On the other hand, the scheme is only computationally concealing, since finding b from the blob would involve determining whether the blob was a quadratic residue or a nonresidue without knowing the factorization of n. An omnipotent verifier could do this, so the verifier must be restricted to normal computational power to use this scheme [12].

It is also possible to have a computationally binding but unconditionally concealing scheme. Consider the following protocol from [3, 12], based on discrete logarithms, in which we have a public prime p, a public generator α of the finite field $\mathbf{Z}_p^*$, and β, which is an element of this finite field chosen by the verifier:

1. The prover encrypts the bit i by choosing a random x and computing the blob $\beta^i \alpha^x$.

2. The prover opens the blob when the time comes simply by revealing i and x.

The verifier chooses β, and he can choose it to be whatever he likes. In particular, he can choose it in such a way that he knows its discrete logarithm l. This means that whatever bit the prover encrypts, the verifier will be able to open it, but he will also be able to open it as the opposite bit: $\alpha^{li}\alpha^{x} \equiv \alpha^{l(1-i)}\alpha^{xl(i-1)} \pmod{p}$. The blobs are thus unconditionally concealing, as without knowledge of x, the verifier will have no way of knowing which bit was encrypted. However, since the blobs can be opened both ways, we need to restrict the prover to conventional computing ability so that she cannot take advantage of this fact. These computational assumptions obviously impact the computational assumptions we can make regarding the participants (limited prover, omnipotent verifier), and also impacts the computational assumptions to be made on the zero-knowledge properties of the proof in which it is used, as we shall shortly see.

2.3 Proofs vs. Arguments

The variant of zero-knowledge proofs in which we assume a verifier with unlimited computational power, and a more limited prover is called a "zero-knowledge argument". The idea was first discussed in [3], and formalized in [5]. Somewhat confusingly, arguments are considered a subset of zero-knowledge proofs while also considered somehow distinct from them. The main difference between arguments and proofs lies in the bit-commitment schemes. The soundness of a zero-knowledge proof using bit commitment is dependent on the binding properties of the bit commitment scheme. If the prover can open a blob as a zero or a one, she can cheat the verifier. Thus, a zero-knowledge proof, with its computationally powerful prover, requires unconditionally binding bit commitment. On the other hand, a zero-knowledge argument requires unconditionally concealing bit-commitment, to keep the verifier from opening the blobs himself. The blobs are now computationally binding, but a computationally limited prover can no longer exploit this property, so the argument is computationally sound. Completeness is of course unconditional for both proofs and arguments [12].

However, the most interesting difference between the two lies in the zero-knowledge property itself. Consider the zero-knowledge properties of the Hamiltonian Cycles protocol if formulated as a zero-knowledge proof. A falsified transcript, whether or not the possibility of a cheating verifier is taken into account, will contain encrypted and permuted instances of a graph H which is not isomorphic to the original graph G. An omnipotent third party will be able to break the computationally concealing blobs, and determine that the graphs are not isomorphic. Thus, this third party will be able to tell a forged transcript from a real one, and the proof achieves only computational zero-knowledge.

On the other hand, if the blobs are unconditionally concealing, and the protocol therefore a zero-knowledge argument, an omnipotent third party will be just as likely to decrypt any given blob as a zero or as a one. Since the unknown quantity that the prover uses to encrypt her bits is chosen randomly, and since the range of blobs generated by zero is identical to that generated by

one, any blob is just as likely to appear in any given location as any other. The result is perfect zero-knowledge.

3 Zero-Knowledge Proofs: Relevant Results

There are several results outside the scope of basic zero-knowledge that are relevant here. Not the least of these is Blum's original zero-knowledge proof for proving theorems. I present the theorem and its proof outline exactly as they appear in [2]:

> Given any logical proof system (such as Russell and Whitehead's extremely general system within which it is generally acknowledged that all mathematical theorems can be formulated and proved), given any theorem provable in that system, and given an upper bound, L, on the length of some proof of the theorem in the system, it is possible to efficiently transform that proof into a zero-knowledge proof of the theorem. This is an interactive probabilistic protocol whereby the prover persuades the verifier with high probability,
>
> (1) the theorem has a proof in the given proof system of length $\leq L$, and
>
> (2) the prover knows such a proof. The probability that a cheater, i.e., a prover who does not know a proof, will pass this test $\leq 1/2^k$ for a protocol with k rounds.
>
> IDEA OF PROOF. The proof system is defined by a nondeterministic TM (Turing machine) which, on input (statement of theorem, 1^n), guesses a proof of the theorem of length $\leq n$, checks if it is a valid proof within the system, and accepts if it is, rejects if not.
>
> The prover gives the verifier a zero-knowledge proof that he, the prover, knows an accepting path for this TM for some n. The protocol for this is along the same lines as for Hamilton Cycle in a graph: one splits the computations into two pieces. The integer n must be chosen by the prover to be an upper bound on the length of the proof in the system. Q.E.D.

3.1 Finite State Machines

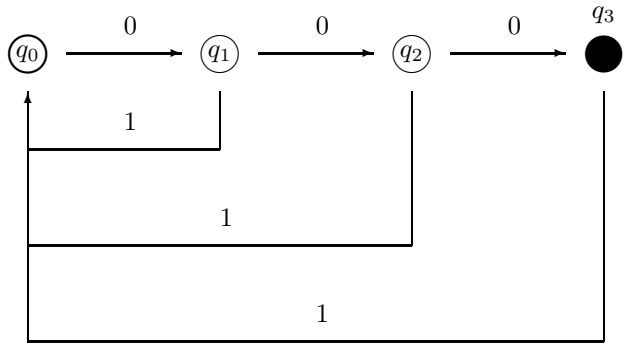
Blum intended the protocol to run in the context of a nondeterministic Turing machine. However, given that it involves keeping track of accepting paths, it is more appropriate to view it in the context of a finite state machine. We would, of course, like to extend these results to Turing machines.

A finite state machine is a computing model that consists of an input tape, a collection of states, and transitions between those states. A state is simply

a configuration of a computer's memory. Any program running on a computer with a finite amount of memory can be modeled as a finite state machine. For example, a computer with sixteen bits of memory has 2^{16} possible states. The particular instance of the problem being solved is encoded on the input tape. The machine reads the input tape in one direction, and can transit between states depending on what symbol it reads. There exists at least one state called an "accepting state". If the machine reaches the end of the input tape while in an accepting state, then the input is a yes-instance of whatever problem is being solved, and the machine accepts. Otherwise, it rejects.

The state machine can be represented as a directed graph, with states as vertices and transitions as edges. For an example, consider the finite state machine illustrated in figure 1. Assume the input tape is written in binary notation. This finite state machine starts out in its initial state q_0, represented by the circle with the thick border. No transition is indicated for any symbol other than a zero, so it simply stays in the initial state if a zero is not read. If a zero is on the input tape, it transits to state q_1. Either way, the read head of the machine advances to the next symbol on the tape. If having advanced to any state q_n other than the initial state, the machine reads a one from the tape, then it returns to the initial state, otherwise it continues to q_{n+1} until it reaches the accepting state q_3, represented by the filled circle. If after this point no more ones are read, then the machine accepts. So, we can see that the machine accepts if and only if the input tape contains a binary representation of an integer divisible by eight.

Figure 1: A finite state machine that determines divisibility by eight.



3.2 Subprotocols

In order to demonstrate a theorem in zero-knowledge, a preliminary zero-knowledge result must first be established. We need a means of demonstrating knowledge of a path P between two vertices a and b in a directed graph. This zero-knowledge proof runs along very similar lines to the Hamiltonian proof shown above.

1. The prover applies a random permutation ϕ to the vertices in the graph G. She encrypts each entry of the adjacency matrix of $\phi(G)$ with a bit-commitment scheme and transmits it to the verifier.

2. The verifier determines a random bit i.

3. If $i = 0$, then the prover reveals the permutation, i.e. opens each and every blob in the adjacency matrix, enabling the verifier to see for himself that $\phi(G)$ is isomorphic to G. If $i = 1$, the prover reveals the path $\phi(P)$. This entails opening the blobs that correspond to the component edges of the path.

4. This procedure is repeated for a total of k rounds.

If the prover does indeed know such a path, the verifier will be convinced of this fact with probability 1. If the prover does not know such a path, she will be able to falsely convince the verifier that she does with probability $1/2^k$. Thus, the protocol is both complete and sound.

The zero-knowledge properties follow from the fact that just as with the proof for Hamiltonian cycles or any other zero-knowledge proof, the verifier never sees the whole picture at once. In any round, the prover will provide the verifier with $\phi(G)$, which may or may not contain the relevant path P, or a path, just like any other path, that may or may not be P, and does not even necessarily come from G. Thus, the verifier can set up a simulator. He simulates the prover by determining a random bit h. If $h = 0$, he permutes G randomly. If $h = 1$, he permutes a graph of size $|G|$ containing a random path of length $|P|$. He then encrypts the adjacency matrix and transmits it to the simulated verifier. The simulated verifier determines a random bit i. If $h = i$, then the simulated prover opens the relevant blobs to the simulated verifier, and the round concludes successfully and is recorded on the transcript. If $h \neq i$, then the simulator resets to the beginning of the round, and starts again. Since any path of length $|P|$ is just as likely as any other such path, and any permutation ϕ is also just as likely as any other, the decrypted portion of the set of simulated transcripts will have the same probability distribution as the decrypted portion of the set of real ones. The proof for the case involving the cheating verifier also proceeds along the lines of that for the Hamiltonian Cycle protocol. Therefore, this protocol is either perfect or computational zero-knowledge, depending on the properties of the bit-commitment scheme employed.

4 The Propositional Calculus $\mathbf{P}_{LT'}$

The setting we have chosen for the theorem decidability problem is the propositional calculus due to Łukasiewicz, given in [10] as $\mathbf{P}_{LT'}$. It is a full classical axiomatic system with negation. Its major distinguishing properties are that its axioms have been pared down to a completely independent set, as have its improper symbols.

4.1 Rules of Formation and Rules of Transformation

The symbols of the propositional calculus $\mathbf{P}_{LT'}$ are partitioned into the set of proper symbols and the set of improper symbols. The set of proper symbols comprises an infinite number of propositional variables, here designated v_p for some integer p. There are four improper symbols: The unary operator $\neg$, the binary operator $\Rightarrow$, and the bracketing symbols (and). The operators are defined in tabular form:

p	q	$p \Rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

p	$\neg p$
T	F
F	T

Thus, these two symbols correspond to the notions of "implication" and "negation", respectively.

A formula is defined as any combination of symbols. A well-formed formula is simply a sequence of symbols put together in such a way as to be meaningful within the rules set down by the given calculus. Within $\mathbf{P}_{LT'}$, the following rules of formation apply:

1. Any proper symbol v_p is a well-formed formula unto itself.

2. If A is a well-formed formula, then its negation $\neg A$ is a well-formed formula.

3. If A and B are well-formed formulae, then the statement that A implies B, enclosed in brackets, is a well-formed formula: $(A \Rightarrow B)$.

Finally, there are two rules of transformation. These rules govern the determination of "immediate consequences" in the system $\mathbf{P}_{LT'}$. The first of these is the rule of substitution: If A and B are well-formed formulae, then B is an immediate consequence of A by substitution if and only if it can be derived by substituting some well-formed formula C for every instance of some variable p

in A.

The second is the rule of modus ponens. If A, B, and C are well-formed formulae, then C is the immediate consequence of A and B by modus ponens if and only if the formula B states "$A \Rightarrow C$".

Of course, for a formula to be considered valid in $\mathbf{P}_{LT'}$ it is not sufficient for it to be derivable from other well-formed formulae. It must be the final link of a chain of implication stretching back to the basic axioms of the calculus.

4.2 Axioms and Theorems

The basis of all deductions in $\mathbf{P}_{LT'}$ is its three axioms:

1. $(v_0 \Rightarrow (v_1 \Rightarrow v_0))$
2. $((v_0 \Rightarrow (v_1 \Rightarrow v_2)) \Rightarrow ((v_0 \Rightarrow v_1) \Rightarrow (v_0 \Rightarrow v_2)))$
3. $((\neg v_0 \Rightarrow \neg v_1) \Rightarrow (v_1 \Rightarrow v_0))$

A theorem is simply a sequence of formulae in which each formula is either an axiom or the immediate consequence of some previous formula or formulae by either of the rules of transformation.

5 Proving a Theorem in $\mathbf{P}_{LT'}$ in Zero-Knowledge

Since the propositional calculus seeks to reduce the exercise of logic to a matter of adherence to simple formal structures related by a small number of rules, it is admirably fitted to the finite state machine model of computation.

5.1 A Finite State Machine for Provability Testing

If a computer program could be written to verify a proof, then such a program could be simulated by a finite state machine. Such a machine would have a starting state and an accepting state. Thus, the knowledge of a proof for a certain formula f is equivalent to knowledge of a path from the initial state, through the states representing f as it is parsed by the program, to the accepting state.

Such a machine would be known to any potential verifiers and third parties in a zero-knowledge proof. A verifier would be able to examine this machine and verify that it does what it is purported to do before any protocol began.

To demonstrate that such a finite state machine exists, I have created a proof-checking program. This program, written in CMU Common LISP, checks proofs given in the form of a list of well-formed formulae. The formulae themselves are denoted as follows: $(A \Rightarrow B)$ is given as $(A \quad B)$, and $\neg A$ is given as (A) The formulae are arranged in "reverse order", i.e., formulae precede their justifying arguments in the list. The program checks each formula successively to verify that it is either an axiom or that it can be derived by a formula or formulae occuring later in the list by substitution or modus ponens. For example, consider the following theorem, which proves the formula $((p \Rightarrow q) \Rightarrow (p \Rightarrow p))$:

1. $(p \Rightarrow (q \Rightarrow p))$

2. $((p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r)))$

3. $((p \Rightarrow (q \Rightarrow p)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow p)))$

4. $((p \Rightarrow q) \Rightarrow (p \Rightarrow p))$

When the proof-checker receives this theorem as input in the form `'(((p q) (p p)) ((p (q p)) ((p q) (p p))) ((p (q r)) ((p q) (p r))) (p (q p)))`, it returns `t`, because formula 4 is derivable from formulae 3 and 1 by modus ponens, formula 3 is derivable from formula 2 by substituting p for each instance of r, and formulae 1 and 2 are axioms. On the other hand, the proof-checker returns `nil` for the following "theorem":

1. $(p \Rightarrow (q \Rightarrow p))$

2. $((p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r)))$

3. $(p \Rightarrow q)$

4. $(q \Rightarrow (p \Rightarrow q))$

This is because no justification is given for formula 3, in spite of the fact that formula 4, the very formula that the "theorem" purports to prove, is itself an axiom and needs no justification. Thus, the proof-checker can be seen to verify the validity of proofs, while not considering the validity of individual formulae independent of the proofs in which they occur. In other words, the program checks proofs, it does not write them.

5.2 Proving a Theorem in Zero-Knowledge

We now have a setting for our zero-knowledge proof. It is a simple application of the proof of the knowledge of a path as presented above: The prover randomly permutes the graph G_t representing the finite state machine, and sends it to the verifier. The verifier determines the random bit i and sends it back to the prover. Finally, if $i = 0$, the prover reveals the permutation, and if $i = 1$, the prover reveals the path P_f from the initial state to the accepting state representing the formula.

Revealing the graph resulting from diagramming the finite state machine is a necessary step. The verifier needs to know that the proof is taking place in the setting of the correct finite state machine. However, as far as the zero-knowledge aspect of the proof is concerned, the fact that a verifier has an unobstructed view of the graph is not a problem. The verifier could just as profitably occupy his time with attempting to find possible predecessors to the formula which is to be proved on his own, without any help from the prover and her finite state

machine. Finding the proof in this manner is no more difficult than finding the proof any other way. And herein lies the problem.

Through a simple breadth-first search, it is possible to determine in time linear in the number of edges in G_t whether a path between any two given points exists. Now, the number of states in any proof-checking finite state machine will be enormous— it will be exponential in both the length of the proof and the maximum allowed formula length, and consequentially, the number of edges in the resulting graph will also be very large. However, it is the graph that the verifier sees. So, while the verifier will indeed gain no knowledge of the theorem proof from executing the protocol, he will be able to find a path from the initial state to the accepting state— and hence the proof— with even more efficiency than the protocol affords him.

6 Conclusion

It still may be possible to create a zero-knowledge protocol to demonstrate knowledge of a proof, but it would have to be based upon a different, intractable problem. Finding such a problem is an area for future work. Once that has been done, the task of generalizing the results to any formal system, such as that of Principia Mathematica, can be undertaken, as was Blum's original intent.

However, as it now stands, the protocol cannot effectively accomplish its goal. Finding a proof for a theorem may or may not be an intractable problem, but any and all intractability vanishes when the finite state machine's graph is presented to the verifier. While a zero-knowledge proof can be seen to exist for this problem, it is essentially a trivial protocol.

References

[1] Blum, M. "Coin Flipping by Telephone: A Protocol for Solving Impossible Problems", Proceedings of the 24th IEEE Computer Conference (CompCon), 1982, pp. 133-137

[2] Blum, M. How to Prove a Theorem So No One Else Can Claim It, Proceedings of the International Congress of Mathematicians, Berkeley, CA, 1986, pp. 1444-1451

[3] Brassard, G., D. Chaum, C. Crepeau, "Minimum Disclosure Proofs of Knowledge", Journal of Computer and System Sciences 37, 1988, pp. 156-189

[4] Brassard, G., C. Crepeau, "Zero-Knowledge Simulation of Boolean Circuits", Advances in Cryptology - CRYPTO '86 Proceedings, Springer-Verlag, 1987, pp. 223-233

[5] Brassard, G., C. Crepeau, "Sorting Out Zero-Knowledge", Advances in Cryptology - EUROCRYPT '89 Proceedings, Springer-Verlag, 1990, pp. 181-191

[6] Galil, Z., S. Haber, M. Yung, "A Private Interactive Test of a Boolean Predicate and Minimum-Knowledge Public Key Cryptosystems", Proceedings of the 26th IEEE Symposium on Foundations of Computer Science, 1985, pp. 360-371

[7] Goldreich, O., S. Micali, A. Widgerson, How to Prove All NP Statements in Zero-Knowledge and a Methodology of Cryptographic Protocol Design, Advances in Cryptology - CRYPTO 86 Proceedings, Springer-Verlag, 1987, pp. 171-185

[8] Goldreich, O., S. Micali, A. Widgerson, "Proofs that Yield Nothing but their Validity or All Languages in NP have Zero-Knowledge Proofs", Journal of the Association for Computing Machinery, 38(1), 1991, pp. 691-729

[9] Goldwasser, S., S. Micali, C. Rackoff, "The Knowledge Complexity of Interactive Systems", SIAM Journal on Computing, 18(1), 1989, pp. 186-208

[10] Hackstaff, L.H. Systems of Formal Logic, Gordon and Breach, New York, 1966

[11] Schneier, B. Applied Cryptography, 2nd ed., Wiley, New York, 1996

[12] Stinson, D.R., "Cryptography– Theory and Practice", CRC Press, New York, 1995