2011

Zero Knowledge Protocols

Caitlin Bonnar
University of Redlands

ZERO KNOWLEDGE PROTOCOLS

BY CAITLIN BONNAR

SPRING 2011

Table of Contents

I. Introduction

In this day and age, it is commonplace to spend part of our day on the Internet. Whether to check e-mail, purchase goods, manage a bank account, or merely browse interesting sites, we rely on certain security measures to keep personal information safe from unwanted outsiders. Within the field of cryptography there are many techniques and algorithms that have provided top-notch security for our methods of communication today, yet as technology advances and as loopholes are found, we are constantly looking for novel ways to protect our information. Introduced approximately 25 years ago by Goldwasser, Micali, and Rackoff [7], zero knowledge protocols seek to do just that. This paper will explore these protocols, their application to *NP*-complete problems (problems with no efficient way of finding a solution), and their use in modern day cryptosystems.

Informally stated, a zero knowledge protocol (abbreviated ZKP from here on) is an interactive method between two parties that allows one party (the *prover*) to prove to the other party (the *verifier*) the veracity of a statement without revealing anything about the statement itself. In other words, 'zero knowledge' is leaked to the verifier other than knowledge of whether the statement is true or not. Note that in the case of ZKPs, the usage of the word "prove" is different from its strict mathematical meaning—in this case, it refers to an overwhelming probability (one that can be made arbitrarily close to 1) that the prover is not cheating the verifier. For example, assume that someone knows the location of a secret treasure chest. They would like to convince someone else that they truly know where this treasure chest is without revealing anything about its location, in order to make sure they will not try and steal it. Therefore the two come up with an

interactive procedure that will allow the prover to prove only the fact that they know this secret knowledge and nothing else. If they accomplish this goal, the procedure is known as a ZKP.

II. Zero Knowledge Protocols

ZKPs rely on three mathematical properties, which will be stated intuitively below and formalized later:

• the *completeness property*: the protocol should allow an honest prover to be able to prove to the verifier that they know what they claim to know with probability 1; that is, an honest prover should succeed every time,

• the *soundness property*: the probability that a dishonest prover who does *not* hold the secret knowledge can deceive the verifier can be made arbitrarily close to 0,

and • the *zero knowledge property*: no information about the prover's secret knowledge is learned by the verifier during the process.

1. A Simple Example

To illustrate these properties, consider the following classic example, introduced by Neal Koblitz in 1994 [10]:

Suppose that Peggy (our prover) knows the secret code to a magical cave door. Victor (our verifier) wishes to purchase this code from her, but he wants to make sure that she is indeed telling the truth before he pays her for it. On the other hand, Peggy does not wish to reveal the secret code to Victor until he has paid her, for fear of not

receiving payment once he learns it. Say that our cave consists of a circular tunnel consisting of two paths, *A* and *B*, which are separated by the magical door, as depicted below:

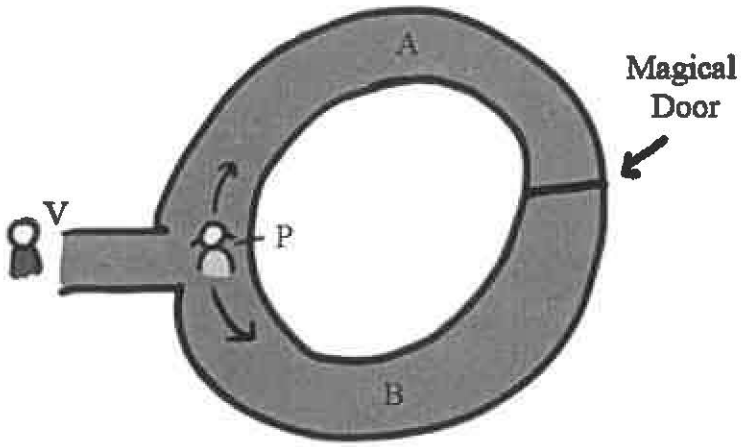

Figure 2.1: A magical cave

The two decide to solve this problem by executing the following protocol:

Protocol 2.1:

The following steps are repeated *k* times, where *k* is a positive integer specified by Victor.

1) Peggy chooses at random to travel down either path *A* or path *B* while Victor waits outside so that he does not see the path Peggy travels down, as shown in Figure 2.1.

2) Standing at the cave entrance, Victor calls out a random path on which he wishes Peggy to exit.

3) Since Victor is now inside the cave and able to see which path Peggy exits from, if he sees her exit from the path he called out, he accepts and continues. Otherwise, Victor rejects Peggy's proof and the protocol ends.

We can see that if Peggy initially chose path A to travel down and Victor asked her to come out from path B, she would have to use the secret code to open the door. Of course, there is a 50% chance that both Peggy and Victor will choose the same path in a single round, but the overall probability that they will choose the same path *every* round is $\left(\frac{1}{2}\right)^k = \frac{1}{2^k}$, which becomes extremely small with a large value k. Therefore, if Victor is satisfied with a 1 in 512 chance that Peggy could be cheating, they will repeat the steps nine times before Victor accepts Peggy's proof that she knows the secret code.

It is simple to show that this protocol satisfies all three properties described above. Peggy should be able to exit down the path Victor calls out each round without fail since she knows the secret code (thus this protocol is complete), the chance that someone else could exit from the requested path each time without knowing the code can be made arbitrarily close to zero as k gets large (thus the protocol is sound), and finally no information about the code is leaked since Victor is outside the cave and cannot see or hear Peggy when she opens the door (thus it is zero knowledge).

2. Formal Definitions

Using these ideas, we will now formalize our definitions.

Definition 2.1 *(Completeness Property)*: An interactive proof protocol is said to be *complete* if, given an honest prover and an honest verifier, the prover convinces the verifier to accept the proof with probability 1.

Definition 2.2 *(Soundness Property)*: An interactive proof protocol is said to be *sound* if, given a dishonest prover (that is, a prover who does not actually know what they claim to know) and a positive integer k chosen by the verifier as a security parameter, the verifier rejects the proof with probability at least $1 - \frac{1}{2^k}$.

Definition 2.3 *(Zero Knowledge Property)*: An interactive proof protocol is said to hold the *zero knowledge property* if there exists a method or procedure that produces a transcript of the proof made with a dishonest prover that is indistinguishable from a real transcript made with an honest prover, called a *simulator* [14, pp. 5-6].

A proof holding the first two properties is referred to as a *proof of knowledge*. Together with the third property, the proof is known as a *zero knowledge proof.*

The only definition that does not seem to fit with our intuitive definitions given earlier is our definition of the zero knowledge property. However, we can think of it in the following manner, borrowing again from our example: Suppose that Victor were to videotape an execution of the protocol from his point of view (that is, from the verifier's point of view) with a prover that does *not* actually know the secret password to the cave door. The two can predetermine the path that Victor will call out each round so that the dishonest prover can travel down this predetermined path each time and never actually have to use the secret code to open the door. If Victor were to then show another party this tape along with a videotape of the honest protocol execution between him and Peggy,

the third party would not be able to determine which tape was the honest one. Since Victor was able to simulate a successful execution of this protocol without an honest prover, we can see that whatever information that is learned by watching the honest interaction with Peggy can also be extracted from an interaction *without* an honest prover, meaning that the information learned during the protocol's execution does not have anything to do with the secret information. Thus the verifier learns nothing about the secret information, which implies our informal definition of zero knowledge given earlier [12, pp. 2-3].

III. ZKP for Hamiltonian Cycles

To provide a more detailed example, we will produce a ZKP that proves whether or not a person knows of a Hamiltonian cycle within a graph. However, before we are able to define what a Hamiltonian cycle is and specify our protocol, we must introduce some basic definitions and concepts from elementary graph theory. We start with the definition of a graph.

Definition 3.1*:* A *graph G* is a diagram consisting of points, or *vertices*, joined together by lines, called *edges*. Each edge either *joins* exactly two vertices or it connects a vertex to itself; the latter is known as a *loop* [2, p. 26].

The above definition is the more intuitive way we think about graphs; we could also say that a graph consists of two sets: a set of vertices and a set of edges, where each edge is labeled vw for the vertices v and w that they connect (for a loop, $v = w$). For this

paper, however, we will use the above pictorial definition. Next, we make the idea of "touching" in a graph more precise:

Definition 3.2: Let v and w be vertices of a graph. Then v and w are *adjacent* if they are joined by an edge e. In addition, we say that v and w are *incident* with the edge e, and e is *incident* with vertices v and w.

Definition 3.3: In a graph, two or more edges joining the same pair of vertices are called *multiple edges*. A graph with no multiple edges or loops is called a *simple graph* [1, pp. 26-27].

Figure 3.1: A loop at vertex a

Figure 3.2: Multiple edges joining a and b

Definition 3.4: A graph G is *planar* if it can be drawn in the plane in such a way that no two edges meet except at a vertex with which they are both incident. Any such drawing is said to be a *plane drawing* of G. A graph is *non-planar* if no plane drawing of G exists [2, p. 244].

Now, take the simple graph below:

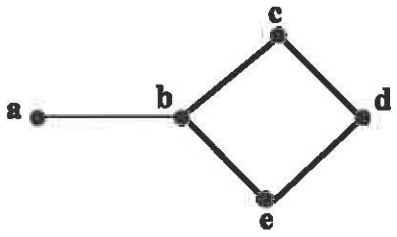

Figure 3.3

Suppose this graph represents a network of cities connected by roads, and suppose Bob is currently in the city represented by the vertex *a*. Bob would like to make his way to the city represented by *d* through the cities and roads depicted in this graph, and so he travels from *a* to *b* to *c* and finally to *d*.

Definition 3.5: A *walk of length k* in a graph is a succession of *k* edges of the form: *uv, vw, wx,...,yz*. This is denoted by *uvwx...yz* and is referred to as a *walk between u and z* [2, p. 39].

In this example, Bob took a walk between *a* and *d*, in particular *abcd*. Note that another walk from *a* to *d* could have been *abed* or even *abcdebcd*. In addition, we can apply the following definition:

Definition 3.6: A *path* is a walk in which all edges and vertices visited are distinct [2, p. 40].

In this case, both *abcd* and *abde* are paths, since Bob visited each city and traveled down each road in the walk only once. However, the walk *abcdebcd* is not considered a path since *b, c,* and *d* are all visited twice.

Definition 3.7: A *closed walk* in a graph is a succession of edges of the form *uv, vw, wx,...,yz, zu,* that starts and ends at the same vertex. A *cycle* is a closed walk in which all edges traveled are distinct and all intermediate vertices are distinct [2, p. 42].

Therefore, continuing with our example, if Bob takes the path a*bcdeba* in order to return to the same city he started out in, we say that he took a closed walk. However, there is no cycle starting at vertex *a*, since we would have to travel back down the edge

joining *a* and *b* in order to return to *a*, and thus all edges traveled would not be distinct. The closed walk *bcdeb is* a cycle however, since every edge traveled is distinct as are the intermediate vertices.

We can now use the above concepts to define more complex forms of graphs.

Definition 3.8: A graph *G* is *connected* if there exists a path between each pair of vertices of *G*. We say a graph is *disconnected* otherwise. Every disconnected graph *H* can be split up into a number of connected subgraphs, that is, smaller graphs within *H*, called *components* [2, p. 41].

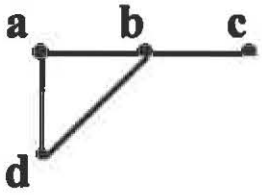

Figure 3.4

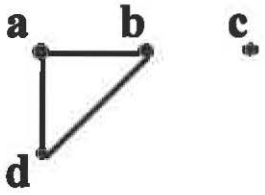

Figure 3.5

For example, the graph in Figure 3.4 is an example of a connected graph because there exists at least one path between each pair of vertices. However, if we remove the edge from *b* to *c* as depicted in Figure 3.5, there is no longer a path from vertex *a* to *c* (nor from any other vertex to *c*) and thus it becomes a disconnected graph with two components, the subgraph *c* and the subgraph *abd*.

Definition 3.9: A *cycle graph* is a connected graph consisting of a single cycle of vertices and edges. A cycle graph with n vertices will be denoted by C_n.

Figure 3.6: Cycle graphs for $n = 1, 2, ..., 6$ [2, pp. 45-46].

These graphs are a subset of the set of Hamiltonian graphs, which we will now define.

Definition 3.10: A *Hamiltonian graph* is a connected graph in which there exists a cycle passing through every vertex [2, p. 71]. We call such a cycle a *Hamiltonian cycle*.

The graph C_n is obviously Hamiltonian for all n by definition, and we can easily see the cycle that passes through each vertex in the examples depicted in Figure 3.6. However, we cannot easily classify most graphs as Hamiltonian or not. Take, for example, the labeled graph of the planar dodecahedron.

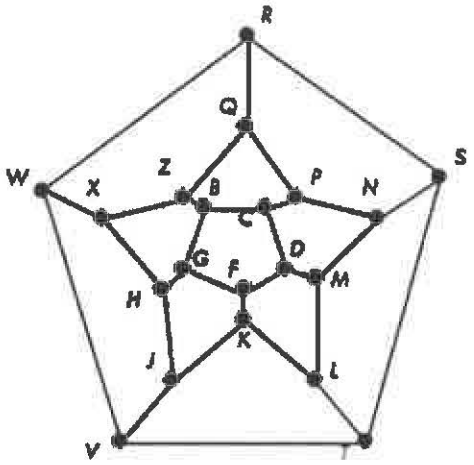

Figure 3.7: A planar dodecahedron

This graph was used by Sir William Rowan Hamilton in a game he invented under the title *A voyage round the world,* in which the labels on the vertices represented places such as Brussels, Canton, and Delhi. In this game, he challenged players to find Hamiltonian cycles starting with five given letters. For example, if the five letters given were *BCPNM*, there are exactly two ways of completing a Hamiltonian cycle: i) *BCPNMDFKLTSRQZXWVJHGB* and ii) *BCPNMDFGHXWVJKLTSRQZB* [2, p. 71].

Even though there are many Hamiltonian cycles in this graph, we can see that it is not as easily classifiable as Hamiltonian as C_n and also that it can take some work to find *all* the Hamiltonian cycles starting with five given letters. Determining whether or a not a graph is Hamiltonian becomes an incredibly difficult problem the more large and complex a graph becomes; in fact, there are no known efficient algorithms that allow us to find a Hamiltonian cycle in a large graph today. Therefore we are left with an exhaustive search method, which becomes more and more time-consuming as the number of vertices in a graph increases.

Finally, we present one more definition before presenting our protocol for knowing of a Hamiltonian cycle in a graph.

Definition 3.11: Two graphs G and H are *isomorphic* if there exists a way to re-label the vertices of G and arrive at H. That is, there is a one-to-one correspondence between the vertices of G and those of H such that the number of edges joining each pair of vertices in G is equal to the number of edges joining the corresponding pair of vertices in H. Such a one-to-one correspondence is an *isomorphism* [2, p. 29].

We will now provide a ZKP associated with Hamiltonian cycles.

Suppose that Peggy knows of a Hamiltonian cycle in a large graph G, and she must convince Victor that she knows of this cycle without showing him where it is. They therefore execute the following protocol:

Protocol 3.1:

Input: A graph G in which Peggy knows of a Hamiltonian cycle; Victor also knows G. This protocol repeats m times, where Victor decides upon m:

1) Peggy constructs a graph H that is isomorphic to graph G and sends H to Victor.

2) Victor asks Peggy to either i) find a Hamiltonian cycle in H or ii) to prove that H is isomorphic to G.

We can intuitively see that this satisfies the three properties of a ZKP, because if Peggy truly knows of a Hamiltonian cycle in a graph, it will be simple to find a Hamiltonian cycle in the isomorphic graph, and Victor will not learn anything about the

actual cycle in G; also, since Peggy doesn't know which of the two tasks Victor will require her to perform, she cannot cheat by creating a non-isomorphic graph in which she already knows a Hamiltonian cycle. Therefore she must know of a Hamiltonian cycle in G and will be able to convince Victor of this fact through the protocol, satisfying the first two properties, and in the process Victor will not learn anything about the Hamiltonian cycle in G [13, p. 259].

To prove this more formally, we need to satisfy Definitions 2.1-2.3.

Proof:

Completeness: To prove that Protocol 5.1 is complete, assume that Peggy truly knows of a Hamiltonian cycle in G, and that she and Victor will execute the protocol as directed. In other words, assume Peggy and Victor are both honest. Then Peggy should be able to construct an isomorphic graph G by first permuting the vertices and edges of G and then making sure each pair of vertices of H has the same number of edges incident with it as the corresponding pair in G. Therefore, if Victor asks her to prove that the two graphs are isomorphic, she should be able to do so with probability 1 because she knows the one-to-one correspondence. Suppose instead Victor asks Peggy to find a Hamiltonian cycle in H. Since Peggy knows of the Hamiltonian cycle in G, she should be able to use her one-to-one mapping to construct a Hamiltonian cycle in H with probability 1. Therefore she should be able to satisfy either one of Victor's requests each round with probability 1 and convince Victor to accept her proof of knowledge of a Hamiltonian cycle in G. Thus this protocol is complete.

Soundness: To prove soundness, we must show that if Peggy is dishonest, the protocol will fail with a probability greater than or equal to $1-\frac{1}{2^k}$ for some security parameter k chosen by Victor. In this case, suppose Victor chooses $k = m$. Assume that Peggy does not know a Hamiltonian cycle in G. She can try and fool Victor in one of two ways: either she can provide him with a graph H that is *not* isomorphic to G in which she already knows of a Hamiltonian cycle, or she can construct an isomorphic graph and hope that he asks her to prove that they are isomorphic. In the former case, if he asks her to show him a Hamiltonian cycle in H she will be able to because she gave him a graph in which she already knew of a Hamiltonian cycle, but if he asks her to prove that the two graphs are isomorphic she will not be able to do so and thus he will reject her proof. In the second case, if he asks her to show him a Hamiltonian cycle in H, she will not be able to because she does not know of a cycle in G and thus cannot use her one-to-one mapping in any helpful way. Therefore her chance of success of fooling Victor in any given round is $\frac{1}{2}$, and her chance of success for m iterations becomes $\left(\frac{1}{2}\right)^m = \frac{1}{2^m} = \frac{1}{2^k}$, meaning she will fail with probability $1-\frac{1}{2^k}$. Thus this protocol is sound.

Zero knowledge: Finally, to prove that this protocol is zero knowledge, we must show that a simulator exists. That is, there exists an algorithm to fake the execution of the protocol with a dishonest prover and verifier. In this case, a dishonest prover and Victor could pre-arrange which tasks the prover will perform each round so that they can cheat accordingly and then film the execution, as they did in the cave example in section II. Then, if an outsider were to view this fake execution alongside a transcript (or

video/film) of the execution made with an honest prover, they would not be able to distinguish between the two because they would only see that the prover was able to prove either that G and H were isomorphic or that H contained a Hamiltonian cycle in each case, which does not provide any information about the Hamiltonian cycle in G. Therefore this protocol holds the zero knowledge property, and thus constitutes a ZKP. □

One could argue that the preceding protocol does not seem to be of much practical use, since knowing a Hamiltonian cycle in a large graph does not seem to be related to security in any way. However, the basis of most public and private cryptosystems rely on problems that are not efficiently solved by any of today's technology, such as finding the prime factorization of a very large composite number. Knowing a solution to one of these "hard problems," such as finding a Hamiltonian circuit in a large graph, and then proving knowledge of this solution with a ZKP can be used to prove the identity of a message sender or party trying to enter a secure system, which eliminates the possibility of a malicious party stealing the solution or key and using it to impersonate the honest party.

Now that we have provided an example of a hard problem with an associated ZKP, we would like to more formally describe what is meant by saying that a problem is "hard," and later show that we can find a ZKP associated with each of these problems so that we can implement them in cryptographic schemes.

IV. *NP*-Complete Problems

One of the most important results about ZKPs currently is that every *NP*-complete problem has a ZKP associated with it, which is fundamental for implementing these

problems in zero knowledge cryptographic schemes. The reason the set of *NP*-complete problems are of interest in cryptography is because of the fact that, as of today, these problems have no known efficient algorithm to solve them. As a result, we can use a solution to one of these problems as a private key, since no other party should be able to figure out this solution without great time and effort.

Before getting into the formal definition of an *NP*-complete problem, we will provide a basis of definitions that make up computational complexity theory in order to gain insight into why these problems are not efficiently solvable by computers today.

Definition 4.1: A *decision problem* is a problem that has a binary (yes/no) answer [5, p. 8].

Algorithms solving these problems can be simulated by a Turing machine, a theoretical machine that reads in symbols from an input tape and deciphers them based on a set of rules meant to mimic the logic of a computer. From this machine we formalize the idea of an algorithm as well as the amount of time required to find a solution to a decision problem.

1. Turing Machines

A Turing machine can be formally defined as follows:

Definition 4.2: A one-tape *Turing machine* (*TM*) is a 7-tuple $M = <Q, \Gamma, b, \Sigma, \delta, q_0, F>$ where:

i) Q is a finite, non-empty set of *states*,

ii) Γ is a finite, non-empty set of symbols called the *tape alphabet,*

iii) $b \in \Gamma$ is the *blank symbol* (the only symbol allowed to occur infinitely often on the tape),

iv) $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of *input symbols,*

v) $q_0 \in Q$ is the *initial state,*

vi) $F \subseteq Q$ is the set of *final* or *halting states,*

and vii) δ is the *transition function* specifying what symbol to write to the tape, which direction (left or right) to shift the tape, and which state to move to [8, p. 148].

For example, take the 7-tuple for the 3-state "busy beaver" TM [11]:

Q = { **A, B, C, HALT** }

Γ = { 0, 1 }

b = 0 = "blank"

Σ = { 1 }

q_0 = **A** = initial state

F = {**HALT**}

δ: see state-table in Figure 4.1

State table for 3 state, 2 symbol busy beaver

Tape symbol	Current state A			Current state B			Current state C		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state
0	1	R	B	1	L	A	1	L	B
1	1	L	C	1	R	B	1	R	HALT

Figure 4.1

This particular machine acts on only two symbols, 0 and 1, where 0 is the blank symbol that initially fills every cell on the input tape. The goal of this TM is to simply write the symbol 1 to the tape until the TM produces the **HALT** state in its transition function. Therefore, since the initial state of this machine is **A** and the first symbol read is 0, the TM will write the symbol 1 to the tape, then move the tape to the right, and then transition to state **B**, as given in the table in Figure 4.1. From there, it will read in another 0, and since it is now in state **B**, it will write the symbol 1, then move the tape left, then transition back to state **A**. It will continue reading in symbols and acting according to the transition function until it reaches the **HALT** state, when the procedure is considered complete. At this point there will be six 1's written to the tape. Note that for other Turing machines, it is possible that a **HALT** state (or other final state) will never be reached, in which case the procedure will continue forever in an infinite loop.

If a TM does halt, we would like to determine how long it takes and whether or not the algorithm simulated by it is efficient. We therefore develop a notion of computation time.

Definition 4.3: The *time complexity function f* for an algorithm with input length n expresses the time requirements of the algorithm by giving the largest amount of time $f(n)$ needed by the algorithm to solve a problem instance of size n. We will assume that

the encoding scheme used and the machine on which it is computed are fixed so that this function is well defined.

We will say that a function f is $O(g(n))$ for a function g whenever there exists a positive constant c such that $|f(n)| \leq c|g(n)|$ for all $n \geq 0$.

Definition 4.4: A decision problem X is said to be solvable in *polynomial time* if the time complexity function of the algorithm used to solve X is $O(p(n))$ for some polynomial function p. Any algorithm whose time complexity function cannot be so bounded is called an *exponential time algorithm* [5, p. 6].

We generally think of problems that can be solved in polynomial time as problems that can be solved "efficiently," whereas those solved in exponential time generally have much more time-inefficient solutions.

We will now look at two different types of Turing machines.

Definition 4.5: A *deterministic Turing machine* (*DTM*) is one such that only one action is performed in any given situation determined by its set of rules; that is, only one path is taken for each computation. A *nondeterministic Turing machine (NDTM)* is one that can take more than one computational path simultaneously. This can be thought of as a split into "parallel" TMs, given that these parallel TMs do not communicate with one another [16].

The example of the busy beaver TM above is an example of a deterministic Turing machine, for each cell of the table specifies only one course of action: write the symbol 1, shift one direction, then transition to one other state. By contrast, if at least

one cell in the table specified to write a 1, shift left OR shift right, and then transition to one other state, then this machine would start computing down two separate paths: one path initiated by a left-shift and one initiated by a right-shift. These paths would continue on simultaneously performing the actions specified by the TM's transition functions, possibly splitting again into multiple paths. In this regard, an NDTM can be thought of having computational paths resembling a tree graph. Note that any DTM can also be simulated by an NDTM since it can be thought of as a single branch within the tree.

Furthermore, it is important to point out that an NDTM, as a theoretical machine, is not limited by a finite number of processors, processor speed, memory, or anything else that a physical computer is limited by in its computations. Thus, something that is quickly solved by an NDTM may take years to solve on today's fastest computer.

2. The Classes *P* and *NP*

With these definitions, we can now define the classes *P* and *NP*.

Definition 4.6: *P* is the class of all decision problems that can be solved in polynomial time by a DTM [5, p. 8].

For example, the problem of determining whether or not an integer is prime is in *P* because an algorithm using Fermat's Little Theorem, $a^p \equiv a \pmod{p}$ for an integer a and a prime p, allows for this problem to be solved on a DTM in polynomial time [1].

Definition 4.7: *NP* is the class of all decision problems that can be solved by a nondeterministic Turing machine in polynomial time.

It is important to note that a decision problem that can be solved by an NDTM in polynomial time does not necessarily mean that the problem has a polynomial time algorithm to solve it; in fact, many problems in *NP* do not yet have any such algorithm to solve them. For example, the problem of whether or not a certain graph contains a Hamiltonian cycle is a problem in *NP* without an efficient algorithm yet to solve it. Since an NDTM allows for an "exhaustive search" method, we can essentially have the NDTM travel down each path containing distinct vertices in the graph until it finds a Hamiltonian cycle or reaches the conclusion that no such path is a Hamiltonian cycle.

As stated earlier, since any DTM can be simulated by an NDTM, then obviously any decision problem that can be solved in polynomial time on a DTM can also be solved in polynomial time on an NDTM, and thus $P \subseteq NP$. However, it has been a much debated question whether or not $NP \subseteq P$, which would imply that $P = NP$. The consequences of $P = NP$ are important, for if it turns out that this is true, then every decision problem in *NP* can be solved in polynomial time by a DTM, and thus there exist efficient algorithms for solving these problems that we have not yet discovered. However, if $P \neq NP$, then the problems in $NP - P$ may not ever have an efficient algorithm to solve them.

Definition 4.8: A decision problem *A* is said to be *reducible* to another decision problem *B* if an algorithm that solves *A* also solves *B*; that is, the answer to *B* is "yes" iff the answer to *A* is "yes."

In other words, if we say that we are reducing a problem to another problem, we mean that we can use the solution of one problem to solve the other problem. Therefore, if we do not have a method to solve problem *A*, but we do have a method for solving

problem B, if there exists a way to reduce A to B then we can use the solution that we compute for B as a solution for problem A. Thus we develop a method for solving problem A.

Take, for example, the simple problem of multiplying. Suppose that we only know how to add, subtract, divide by two, and take squares, but we do not know how to multiply. Therefore we reduce the problem of multiplying to one that we already know:

Equation 4.1: $ab = \frac{(a+b)^2 - a^2 - b^2}{2}$

In this case, our reductions are algebraic, but we can see that our solution to the problem $\frac{(a+b)^2 - a^2 - b^2}{2}$ is the same as the solution to ab. Even though the reductions of decision problems to other decision problems used on Turing machines are much more complex and thus beyond the scope of this paper, the idea remains the same.

3. *NP*-Hard and *NP*-Complete Problems

Definition 4.9: A decision problem is said to be *NP-hard* if it is reducible to any other problem in *NP* [17].

As of today, this is the set of problems that do not yet have an efficient algorithm to solve them with today's computers. Note that it is possible for a problem that is *not* in *NP* to be *NP*-hard. For example, the decision problem known as the halting problem which states, "given a program and its input, will it ever halt?" has been shown to be reducible to any problem in *NP*, but it is not itself in *NP* because it is not generally solvable in a finite number of steps.

Finally, we can define what it means for a problem to be *NP*-complete.

Definition 4.10: A decision problem X is said to be *NP-complete* if $X \in NP$ and X is *NP*-hard.

In other words, if X is *NP*-complete it is at least as difficult to solve as any other problem in *NP* and there is a way to solve it in polynomial time on an NDTM. This gives us a set of the "hardest" problems to solve in *NP*.

Figure 4.2 contains a pictorial representation of the classes we have discussed so far. One should observe that are currently problems in *NP* that are neither in *P* nor *NP*-complete, known as the *NP*-intermediate problems. An example of an *NP*-intermediate problem is the integer factorization problem, which involves finding the prime factors of a large composite integer *n*. This problem is the basis of many cryptographic systems today.

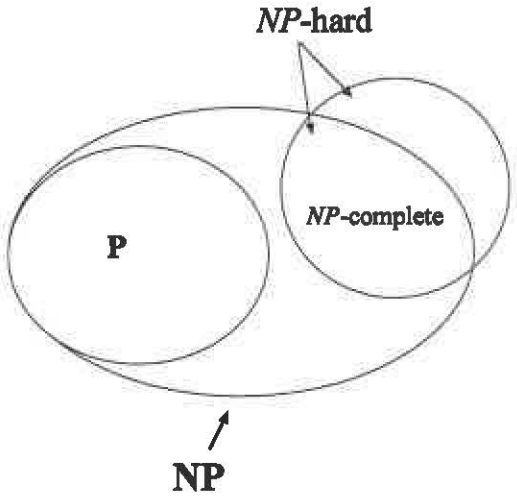

Figure 4.2

The primary impact of the theory of *NP*-completeness on computer scientists is to assist algorithm designers by having them focus on approaches that have the highest probability of producing an efficient algorithm. However, the application of *NP*-complete problems that is of the most interest in this paper, as well as in the field of cryptography, is that because these problems cannot be efficiently solved by any of today's computers, they are valuable in ensuring the security of most cryptosystems.

V. A ZKP for Every *NP*-Complete Problem

Now that we have discussed what *NP*-complete problems are and their role in modern day cryptosystems, we will lead up to and present an outline of the proof that every *NP*-complete problem has a ZKP associated with it. First, however, we must briefly introduce bit commitment schemes, which are used in many ZKPs, including the ZKP given in the upcoming proof.

1. Bit Commitment Schemes

Most modern cryptographic ZKPs implement the notion of *commitment*, which essentially means that once a party in a protocol chooses some value from a finite set, they cannot change their mind from there on out. This is done to enforce honest behavior from each party. A general protocol using a bit (or value) commitment scheme between two parties, say Alice and Bob, occurs as follows: i) Alice commits to a single secret bit and then sends the encrypted or hidden commitment (for example, you could think of it as being locked inside an opaque box) to Bob, ii) Bob commits to a bit and sends it unencrypted to Alice, and then iii) Alice sends a key to Bob that allows him to decrypt or reveal Alice's commitment. There are two essential properties of this scheme as well as any other commitment scheme: that the original value or bit that Alice commits to cannot

change after she sends it to Bob, which is referred to as the *binding property*, and that Bob cannot know Alice's bit until after he commits to his own bit by sending it to Alice, known as the *hiding property*. These schemes are mainly used for hiding or encoding secret information within protocols.

For a simple example, suppose that Alice would like to purchase an item from Bob. They agree that the price that Alice will pay for the item will be $\frac{a+b}{2}$ where a is the price Alice is willing to pay for the item and b is the price that Bob is willing to sell it for. In order to prevent one party from changing their price after hearing the other's offer, they engage in the following protocol:

Protocol 5.1:

i) Alice commits to her price a, writes it on a piece of paper, and sends it in a locked box to Bob.

ii) Bob sends Alice his price, b.

iii) Alice reveals her commitment by giving Bob the key to open the box and view the price.

Note that after Alice sends her price to Bob it cannot change, and Bob will not know Alice's price until after he has sent her his price and she has given him the key; therefore this protocol holds both the binding and hiding property [3, p. 7].

2. A Protocol for the Graph 3-Colorability Problem

In order to show that all *NP*-complete problems are associated with a ZKP, we observe a zero knowledge protocol utilizing a bit commitment scheme for the graph 3-

colorability problem (G3C). This problem can be stated as follows: Given a graph, is there a way to map each vertex to one of three colors in such a way that no two adjacent vertices share the same color? If the answer to this question is 'yes' for a graph G, we say that $G \in$ G3C, because decision problems that can be solved by Turing machines are thought of in terms of sets, where the inputs (or problem instances) that lead to an answer of 'yes' when executed on the TM are in the set, and the problem instances where the answer is 'no' are not.

Since this problem is known to be *NP*-complete [6, p. 99], showing that there exists a ZKP for this problem implies that a ZKP can be constructed for every *NP*-complete problem. This is because each problem that is *NP*-complete is reducible to every other, thus you can reduce any problem that is *NP*-complete to G3C and then execute the ZKP that will be shown to be associated with G3C. Therefore, if you cannot produce a ZKP for any particular problem, this method is always available, albeit impractical.

The following protocol is a proof of knowledge of a 3-coloring in a given graph between a prover P and verifier V:

Protocol 5.2:

Input: A graph G with no loops containing n vertices and m edges, known to both prover and verifier.

The following is executed m^2 times, each time requiring distinct decisions:

1) *P* randomly permutes the set of colors of the 3-coloring, and sends *V n* opaque locked boxes, where the *i*th box contains the color of the *i*th vertex
2) *V* chooses a random edge and sends it to *P*
3) *P* returns keys to the two boxes corresponding to the two vertices incident with this edge
4) *V* opens these boxes, and if the colors are distinct he accepts and continues; if they are the same color, he rejects the proof

If all m^2 iterations are completed successfully, *V* accepts *P*'s proof.

To illustrate this protocol, take the following 3-colored graph *G*:

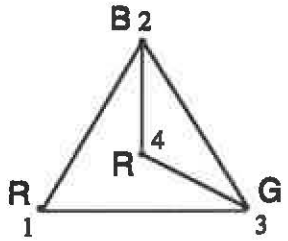

Figure 4.1: A 3-colored graph

Suppose Peggy would like to prove with zero-knowledge to Victor that she knows of this 3-coloring. Therefore the two execute Protocol 4.2 in the following manner:

In the first round, suppose that the random permutation Peggy generates is {R, G, B} → {G, R, B} (that is, red is replaced by green, green by red, and blue remains the same). After applying this permutation, the vertex coloring becomes:

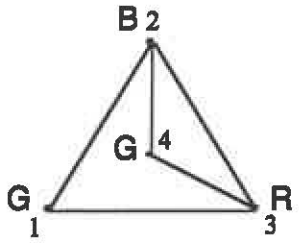

Figure 5.2: *G* after color permutation is applied

Peggy now places each vertex along with its color in its own box, which she locks and sends to Victor. Next, Victor returns the edge (1, 3) to Peggy, and so she sends him the keys to open the boxes containing vertices 1 and 3. Victor opens each box, and seeing that vertex 1 is colored green and vertex 3 is colored red, accepts Peggy's proof this round and continues on to the next round.

In the next round, assume that the permutation generated for the application to the original coloring is {R, G, B} $\rightarrow$ {B, R, G} and the edge selected is (2, 3). After opening, Victor sees that the vertex 2 is colored green and vertex 3 is colored red, so he again accepts and continues.

Suppose now that after this second round Victor wants to try to see if he can put together any information about the original 3-coloring based on the colors he has seen for each vertex. Piecing together the colors he has received for vertices 1, 2, and 3 so far, he arrives at the following coloring:

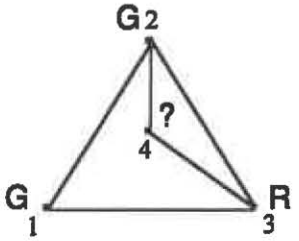

Figure 5.3: Victor's coloring based on the first two rounds

We can see that this is obviously not a valid 3-coloring, since adjacent vertices 1 and 2 are both colored green. This stems from the fact that separate randomly selected permutations are used in between rounds, which is key in describing why this protocol holds the zero-knowledge property.

Peggy and Victor perform 23 more rounds until Victor accepts Peggy's proof and the protocol is complete.

In order to prove that this is a ZKP, we must show that it holds the soundness, completeness, and zero-knowledge properties.

Proof:

Let G be a graph with no loops containing n vertices and m edges.

Completeness: Assume our prover knows of a 3-coloring within a graph. Then any pair of boxes u and v corresponding to some edge of G that the verifier selects will be colored differently even after permuting the colors, according to the definition of a 3-coloring. Therefore, an honest verifier will be able to complete all m^2 iterations and accept with probability 1.

Soundness: Assume our prover does not know of a 3-coloring in a graph. Then at least one of the m edges of G is not properly colored, and thus two adjacent vertices share a color. Therefore, each round has probability at least $\frac{1}{m}$ that the verifier will reject and for m^2 rounds, the total probability of the verifier rejecting is at least $1-\left(1-\frac{1}{m}\right)^{m^2}$. Recall that from our definition of soundness, we need this probability to be greater than or equal to $1-\frac{1}{2^k}$ for some positive integer k chosen as a security parameter. If we let $k=m$, we must show that:

(1) $1-\left(1-\frac{1}{m}\right)^{m^2} \geq 1-\frac{1}{2^m}$, which is satisfied if

(2) $\left(1-\frac{1}{m}\right)^{m^2} = \left[\left(1-\frac{1}{m}\right)^{m}\right]^{m} \leq \left(\frac{1}{2}\right)^{m}$ or, more simply,

(3) $\left(1-\frac{1}{m}\right)^{m} \leq \frac{1}{2}, \forall m \in \mathbf{Z}^{+}$.

If we look at the sequence $\left(1-\frac{1}{m}\right)^{m}$ for $m \geq 1$ we observe that it is strictly increasing by verifying that $\frac{d}{dx}\left(1-\frac{1}{x}\right)^{x} > 0 \Leftrightarrow \frac{d}{dx}\left[x\ln\left(1-\frac{1}{x}\right)\right] > 0$.

Using the product rule and chain rule and then simplifying, we arrive at,

(4) $\frac{d}{dx}\left[x\ln\left(1-\frac{1}{x}\right)\right] = \ln\left(1-\frac{1}{x}\right)+\frac{1}{x-1}$

where, for $x>1$,

(5) $\ln\left(1-\frac{1}{x}\right)>0$ and $\frac{1}{x-1}>0 \Rightarrow \frac{d}{dx}\left[x\ln\left(1-\frac{1}{x}\right)\right]>0$

Next we must check that this sequence converges, and if so, see that it is bounded above by $\frac{1}{2}$. Indeed, we find that:

(6) By definition of e, $e=\lim_{x\to\infty}\left(1+\frac{1}{x}\right)^x$, so if we let $x=\frac{m}{r}$,

(7) $\lim_{m\to\infty}\left(1+\frac{1}{\frac{m}{r}}\right)^{\frac{m}{r}}=\lim_{m\to\infty}\left(1+\frac{r}{m}\right)^{\frac{m}{r}} \Rightarrow \lim_{m\to\infty}\left(1+\frac{r}{m}\right)^{m}=\left[\lim_{m\to\infty}\left(1+\frac{r}{m}\right)^{\frac{m}{r}}\right]^r=e^r.$

Finally, if we let $r = -1$, we have

(8) $\lim_{m\to\infty}\left(1-\frac{1}{m}\right)^m=e^{-1}\approx .3679$, where $.3679 < .5$, and thus we have reached what we were trying to show, and our protocol is sound.

Zero-knowledge: Recall that an interactive proof holds the zero-knowledge property iff there exists a simulator for the protocol. In other words, there exists a way to simulate this protocol in such a way that a fake transcript (for example, the output of a Turing machine) is indistinguishable from a real one.

A machine that represents an honest prover successfully executing this protocol will output only two randomly permuted, distinct colors at the end of each round, for that is all that the verifier is able to see each round. Therefore, if we construct a machine that

merely outputs two random, distinct colors to act as our simulator, we will be left with a transcript that is indistinguishable from a real one, which was our goal. Since each color is equally likely to be an output each round due to the application of random permutations, then probabilistically these two transcripts will be identical. Therefore by definition, our protocol holds the zero-knowledge property, and thus constitutes a zero knowledge protocol for the graph 3-colorability problem (G3C) [12, p. 6-7]. □

We can think of the proof of zero-knowledge more intuitively by thinking about why the verifier does not learn anything about the original 3-coloring of G during execution of Protocol 5.2. One should note that the most important part of this protocol is that the permutation of the colors is applied randomly and independently each round, so that the verifier cannot gather any information about the original 3-coloring. In other words, if the permutation did not change between rounds (and the verifier knew of this), they could decipher the coloring of possibly two pairs of adjacent vertices after only two rounds, depending on the edges chosen. This would be a leak of information and thus violate the zero knowledge property. However, knowing only the coloring of two adjacent vertices each round would *not* allow the verifier to put together any significant information about the original coloring, since it is obvious that the two vertices must be colored distinct colors already.

Finally, we can formalize our main result:

Theorem 5.1: Every *NP*-complete problem has a zero knowledge proof.

Proof (Sketch): Let L be an *NP*-complete decision problem, and let t be an invertible polynomial-time reduction of L to G3C. Since by our definition of reducibility,

the answer to L is yes iff the answer to G3C is yes, we can say that $x \in L$ iff $t(x) \in$ G3C, where x is an input specified by the problem instance of L. (In the case of G3C, it is a graph without loops; this is our *t(x)*.)

Protocol 5.3:

1) Given input x, both prover and verifier compute $t(x) = G$.

2) The prover then uses Protocol 5.2 to show that G is 3-colorable. The verifier accepts proof of knowledge of x based on the acceptance of knowledge of a 3-coloring in G. In other words, the verifier accepts that $x \in L$ based on the fact that $t(x) \in$ G3C.

Since Protocol 4.2 has been shown to constitute a ZKP for G3C, Protocol 5.3 is therefore a ZKP for L. To see that it is indeed zero-knowledge, one should note that since nothing is learned about the 3-coloring of G through Protocol 4.2, nothing is learned about the knowledge of x when G is reduced back to x via $t^{-1}(t(x)) = x$, where we know t^{-1} exists because t is invertible [12, p. 8]. □

Furthermore, it has been shown that a ZKP can be constructed for any problem in *NP* (that is, any decision problem that can be verified in polynomial time on a non-deterministic Turing Machine) [6, p. 184]. These results provide us with a set of problems, many of them inefficient to solve with any current technology, that can be used with ZKPs in cryptographic identification schemes.

VI. Cryptographic Application: Identification

One of the more practical aims of zero knowledge protocols in cryptography is identification, or the act of proving that an individual or party is who they claim to be. For example, take the situation of a user logging into a server. In a typical Unix setup, the user has a password x, and the server holds a hash of the user's password $f(x)$, where f is a one-way function. That is, it is easy to find $f(x)$ given x but infeasible to find x given $f(x)$. To log in, the user will send the server $\bar{x}$, for which the server computes $f(\bar{x})$ and then compares with its stored $f(x)$ in order to grant or deny access to the user. However, this procedure is considered insecure because $\bar{x}$ can be stolen through various means by a hacker or "eavesdropper" and then used to impersonate the user. To create a more secure identification scheme, suppose that a user P picks a random secret key x and then generates a public key $f(x)$, where f is a one-way function. In order for P to prove their identity, they will enter into a ZKP with the server where P acts as the prover and the server acts as the verifier. In this case, P will prove that they know an inverse of $f(x)$, in this case x. This is only possible if P knows x since f is one-way and therefore P cannot derive x from $f(x)$, and since no information about x itself is leaked during the protocol, no eavesdropper can steal x and impersonate P.

In order for any identification scheme to be successful, it must rely on information that is unique to the individual identifying himself. In the above example, this information is a password, which can be created by the user or assigned by the server. In general this information is a secret key that only the user and perhaps a trusted, independent entity (such as a server) knows. In many modern identification schemes, these secret keys are solutions to difficult problems such as *NP*-complete problems. In

fact, many popular cryptosystems, such as RSA, rely on the inefficiency of solving a difficult problem for security purposes. For example, since finding the prime factorization of a large number n is considered to be infeasible with today's technology, using such an n as a modulus ensures that figuring out a secret key that is computed using the prime factors of n is also infeasible.

One such example that relies on the solution to a hard problem is the Feige-Fiat-Shamir proof of identity, introduced in [4]. This scheme is one of the most popular zero knowledge identification schemes today. As in the example of logging into a server given above, the goal of this scheme is to prove the prover's identity to the verifier by showing that they know a secret s that only the prover would know. In this case, the prover has a set of public keys and private keys, as we will see below.

Protocol 6.1: *(The Feige-Fiat-Shamir Proof of Identity)*

Setup: An arbitrator (that is, a trusted independent entity) generates a random number n, the product of two large primes p and q, as a common modulus. Peggy has k secret integer valued keys, $s_1, \ldots, s_k$ such that $\gcd(s_i, n) = 1$ for $1 \leq i \leq k$, and creates $v_i \equiv s_i^{-2} \pmod{n}$ (where we know s_i^{-2} is an integer since s_i is relatively prime to n and thus an integer inverse s_i^{-1} exists modulo n). Each v_i is sent to Victor, or in other words, the v_i's are Peggy's public keys. They then execute the following steps t times, where t is a parameter chosen by Victor, with each round consisting of different random choices.

1) Peggy chooses a random integer r, computes $x \equiv r^2 \pmod{n}$, and sends x to Victor.

2) Victor chooses bits $b_1, \ldots, b_k$ where each $b_i \in \{0,1\}$. He sends these to Peggy.

3) Peggy computes $y \equiv rs_1^{b_1} s_2^{b_2} \cdots s_k^{b_k} \pmod{n}$ and sends y to Victor.

4) Victor then checks that $x \equiv y^2 v_1^{b_1} v_2^{b_2} \cdots v_k^{b_k} \pmod{n}$ and if true, accepts and continues. After t successful iterations, Victor accepts Peggy's proof of identity.

To illustrate why this will work if Peggy truly knows the secret keys, take the case when $k = 1$. Then Peggy is asked for either r or rs_1 depending on whether or not Victor chooses b_1 to be 0 or 1. If he chooses b_1=0, then obviously this will work because $y \equiv r \pmod{n} \Rightarrow y^2 \equiv r^2 \equiv x \pmod{n}$. If b_1=1, then $y \equiv rs_1 \pmod{n}$. If Peggy truly knows s_1, then Victor will be able to successfully compute x because

$$y^2 v_1 \equiv (rs_1)^2 v_1 \equiv r^2 s_1^2 v_1 \equiv r^2 s_1^2 s_1^{-2} \equiv r^2 \equiv x \pmod{n}, \text{ since } v_1 \equiv s_1^{-2} \pmod{n}.$$

Obviously in this case Victor should never choose b_1 to be 0, since Peggy can get away without knowing s_1, so for a more interesting case we will look at when $k > 1$.

To illustrate this case, suppose that Victor sends $b_1 = b_2 = b_4 = 1$ and lets all other $b_i = 0$. Peggy must then produce $y \equiv rs_1 s_2 s_4 \pmod{n}$, which is a square root of $xv_1^{-1} v_2^{-1} v_4^{-1}$ modulo n. In fact, for each round Victor will ask for a square root of the form $rs_{i_1} s_{i_2} \cdots s_{i_j}$, which Peggy will only be able to supply if she knows $r, s_{i_1}, \ldots, s_{i_j}$ since computing such a square root modulo n is just as hard as factoring n itself [15, pp. 81-82].

If Peggy does not know any of the $s_1, \ldots, s_k$, she could guess the string of bits that Victor will send. If she guesses correctly before she sends x, she could let y be a random number and then send $x \equiv y^2 v_1^{b_1} v_2^{b_2} \cdots v_k^{b_k} \pmod{n}$. However, if she guesses incorrectly, she will have to modify her choice of y, which means she will need to compute some of

the square roots of the v_i's, which as we stated before is a difficult task. Of course, her chances of guessing the correct bit string in one round is only $\frac{1}{2^k}$, and for an entire execution of t iterations, her chance of guessing the correct bit string every time is $\frac{1}{2^{kt}}$ and thus the probability that the verifier will reject if she is cheating is $1-\frac{1}{2^{kt}} \geq 1-\frac{1}{2^k}$ which satisfies soundness [15, pp. 231-232].

Note that intuitively this protocol is zero-knowledge because the only values learned by Victor each round are x and y, where x is the square of a random integer unrelated to any s_i and where factoring y into the correct $y \equiv r{s_1}^{b_1}{s_2}^{b_2}\cdots{s_k}^{b_k} \pmod{n}$ is highly infeasible, especially because r is different each round. A video recording in which Peggy and Victor have worked out the bit strings beforehand could also easily simulate this protocol.

The Feige-Fiat-Shamir identification scheme can be implemented in several situations requiring identification, including logging into a server, withdrawing money from an ATM, and proving that you are the party you claim to be in a secure communication. In fact, it is the basis of many cryptographic identification schemes today.

VII. Conclusion

Even though zero knowledge protocols are still a relatively new structure, they already have important applications in the fields of mathematics and computer science. Their use in cryptographic identification schemes such as the Feige-Fiat-Shamir

Identification Scheme not only allows for a person to identify himself, but also guarantees that an eavesdropper cannot steal and use their secret information. Furthermore, the result that every *NP*-complete problem has a ZKP associated with it means that there will always be a set of difficult problems that can be implemented in cryptographic schemes implementing a ZKP.

Finally, one of the more fascinating aspects of zero knowledge protocols is that they are not limited to one particular area of mathematics. Protocols can be designed for problems in graph theory, such as knowing of a Hamiltonian cycle or a graph 3-coloring; they can be designed to test the knowledge of a large square root modulo n; they can even be constructed to prove that you know how to prove a mathematical theorem. The fact that their applicability extends to such diverse fields shows that we may still have many uses for them in times to come.

VIII. References

[1] Agrawal, Manindra and Kayal, Neeraj. "Primes in P." Department of Computer Science and Engineering at the Indian Institute of Technology Kanpur.

http://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf

Proof that an algorithm solving the decision problem of whether or not an integer is prime or not is in the class *P*.

[2] Aldous, Joan M. and Wilson, Robin J. *Graphs and Applications: An Introductory Approach*. London: Springer. 2000.

As the title suggests, provided an introduction to graph theory and served as a basis for most of my graph theory definitions.

[3] Caha, Libor. "Applications of Bit Commitment." [Lecture Slides].

http://www.fi.muni.cz/kd/events/cikhaj-2010-feb/slides/caha_bit_commitment.pdf

Reference for the Protocol 4.1, the bit commitment example.

[4] Feige, Uriel, Fiat, Amos, and Shamir, Adi. "Zero Knowledge Proofs of Identity." *Proceedings of the nineteenth annual ACM symposium on theory of computing*. New York, NY, 1987.

Origin of the Feige-Fiat-Shamir identification scheme.

[5] Garey, Michael R. and Johnson, David S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W.H. Freeman and Company. 1979.

An introductory text to the classes *P* and *NP*, and *NP*-completeness. Provided definitions used in the paper.

[6] Goldreich, Oded, Micali, Silvio, and Wigderson, Avi. "Proofs that Yield Nothing But their Validity and a Methodology of Cryptographic Protocol Design." Paper presented at *Proceedings of the 27th Annual Symposium on Foundations of Computer Science* (SFCS '86), Washington D.C., 1986.

Reference for Protocol 4.2 as well as the proof that all *NP*-complete problems have a ZKP associated with them. (In particular, the proof that Protocol 4.2 holds the zero-knowledge property.)

[7] Goldwasser, Shafi, Micali, Silvio, and Rackoff, Charles. "The Knowledge Complexity of Interactive Proof Systems." *SIAM Journal on Computing* (1989): 291-304.

The article in which zero knowledge protocols were originally introduced and defined. Includes their applications to the complexity class NP, as well as quadratic residues.

[8] Hopcroft, John and Ullman, Jeffrey. *Introduction to Automata Theory, Languages and Computation*. Addison–Wesley, Reading Mass. 1979.

Provided the definition for a Turing machine.

[9] Karp, Richard M. Miller, Raymond E., and Thatcher, James W. "Reducibility Among Combinatorial Problems." *The Journal of Symbolic Logic*, **40**, no. 4. (December 1975).

Reference for proof that the G3C problem is *NP*-complete.

[10] Koblitz, Neal. *A Course in Number Theory and Cryptography*. New York: Springer. 1994.

Source for the magical cave example.

[11] Lin, Shen and Radó, Tibor. "Computer Studies of Turing Machine Problems," *Journal of the ACM*. **12** (1965): 196-212.

Provided the 2-symbol 3-state busy beaver Turing machine example.

[12] Mohr, Austin. "A Survey of Zero-Knowledge Proofs with Applications to Cryptography." Southern Illinois University at Carbondale. http://www.austinmohr.com/Home_files/zkp.pdf

Reference for the proof that Protocol 4.2 is a ZKP.

[13] Mollin, Richard A. *An Introduction to Cryptography*. Florida: Chapman & Hall/CRC. 2001.

Applies ZKPs to Hamiltonian Cycles and other noninteractive protocols.

[14] Simari, Gerardo I. "A Primer on Zero Knowledge Protocols." Universidad Nacional del Sur. http://cs.uns.edu.ar/~gis/publications/zkp-simari2002.pdf

An introductory text to Zero Knowledge Protocols and their applications. Provides a general array of applications of zero knowledge protocols and provides a basis for complexity classes.

[15] Trappe, Wade and Washington, Lawrence C. *Introduction to Cryptography with Coding Theory.* New Jersey: Prentice Hall. 2002.

Used as reference for the Feige-Fiat-Shamir identification scheme.

[16] Weisstein, Eric W. "Nondeterministic Turing Machine." *Wolfram Mathworld.* http://mathworld.wolfram.com/NondeterministicTuringMachine.html

Provided a definition of a nondeterministic Turing machine.

[17] Weisstein, Eric W. "*NP*-Hard Problem." *Wolfram Mathworld.* http://mathworld.wolfram.com/NP-HardProblem.html

Provided a definition of *NP*-hard problems.