Zero Knowledge Compilers

John T. McCall
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
mcca0798@morris.umn.edu

ABSTRACT

Zero knowledge protocols have useful applications in cryptography, however they are usually designed by hand. They can be difficult to design correctly, and it can be easy to make mistakes, as zero knowledge protocols are complex structures. Even if they are designed perfectly, a programmer tasked with implementing it could find themselves in over their head, especially if they lack a deep cryptographic background. The goal behind zero knowledge compilers is to help alleviate these concerns. Using a zero knowledge compiler, one can simply input a proof goal and the compiler will output an implementation of that goal in a high level language, such as Java or C++. The compiler also guarantees correctness of the protocol, which eliminates the risk of subtle mistakes in either the design or the implementation of the protocol.

Keywords

Zero Knowledge Protocols, Compilers, Zero Knowledge Compilers, ZKPDL, ZKCrypt

1. INTRODUCTION

Zero knowledge protocols provide a way of proving that a statement is true without revealing anything other than the correctness of this claim. Zero knowledge protocols have practical applications in cryptography and are used in many applications. While some applications only exist on a specification level, a direction of research has produced real-world applications. One such example is Direct Anonymous Attestation (DAA), a privacy-enhancing mechanism for remote authentication of computing platforms, which has been adopted by the Trusted Computing Group (TCG).

Traditionally, the design of practical zero knowledge protocols is done by hand. Designers use standard arguments and tricks which can be combined and repeated in various combinations to provide the desired, secure, protocol. There are a few problems with this type of method however. The implementations tend to be time-consuming and error-prone. Minor changes in the protocol specification often lead to major changes in the implementation. The protocols are usually designed by cryptographers and implemented by software engineers. The cryptographers typically are not skilled in implementation matters and the software engineers typically have a hard time understanding the complexities of the zero knowledge protocols [3].

Zero knowledge compilers help to alleviate these issues by providing a way to automatically generate zero knowledge protocols for a large class of proof goals. They allow developers to implement these protocols without having an in depth knowledge of cryptography and without having to worry about introducing security flaws in their implementations.

In sections 2 and 3, I provide background on zero knowledge protocols and compilers respectively. In section 4, I talk about some concepts from cryptography that are needed in understanding the compilers. In section 5, I talk about three different implementations of a zero knowledge compiler. Finally, in section 6, I talk about electronic cash, an application of zero knowledge protocols.

UMM CSci Senior Seminar Conference, December 2013 Morris, MN.

2. ZERO KNOWLEDGE PROTOCOLS

Zero knowledge protocols, also referred to as zero knowledge proofs, are a type of protocol in which one party, called the *prover*, tries to convince the other party, called the *verifier*, that a given statement is true. Sometimes the statement is that the prover possesses a particular piece of information. This is a special case of zero knowledge protocol called a *zero knowledge proof of knowledge* [9]. Formally, a zero knowledge proof is a type of interactive proof.

An interactive proof system is an interaction between a *verifier* executing a probabilistic polynomial-time strategy and a *prover* which executes a computationally unbound strategy. satisfying the following properties:

- *Completeness*: If the statement being proved is true, an honest verifier (a verifier correctly following the protocol) will be convinced after interacting with an honest prover.
- *Soundness*: If the statement is false, no prover, either honest or dishonest, will be able to convince an honest verifier, except with a small probability.

For an interactive proof to be a zero knowledge proof it must also satisfy the condition of *zero knowledge*. A proof is zero knowledge if any knowledge known by the prover or the verifier before performing the proof is the same as the knowledge known by either party after performing the proof. In other words, no additional knowledge is gained by either party because of the proof. Another way of thinking about it is that the proof reveals zero knowledge [5].

2.1 Examples

Below are two examples of zero knowledge protocols. The first is a simple example which highlights how a zero knowledge protocol functions. The second is a more practical example, proving knowledge of a Hamiltonian cycle in a graph.

2.1.1 The Magic Cave

The classic example for zero knowledge protocols is the cave example. First presented in [7] and then restated in [5], the cave example is the go-to example for learning zero knowledge protocols.

Peggy has stumbled across a magical cave. Upon entering the cave there are two paths, one leading to the right and one leading to the left. Both paths eventually lead to a dead end, however Peggy has discovered a secret word that opens up a hidden door in the dead end, connecting both paths.

Victor hears about this, and offers to buy the secret from Peggy. Before giving Peggy the money Victor wants to be certain that Peggy actually knows this secret word. How can Peggy (the prover) convince Victor (the verifier) that she knows the word, without revealing what it is?

The two of them come up with the following plan. First, Victor will wait outside the cave while Peggy goes in. She will randomly pick either the right or the left path and go down it. Since Victor was outside he should have no knowledge of which path Peggy took. Then Victor will enter the cave. He will wait by the fork and shout to Peggy which path to return from.

Assuming that Peggy knows the word, she should be able to return down the correct path, regardless of which one she started on. If Victor says to return down the path she started on, she simply walks back. If Victor says to return down the other path, she whispers the magic word, goes through the door, and returns down the other path.

If Peggy doesn't know the word, there is a 50% chance that Victor will choose the path she did not start down. If this happens there is no way that she can return down the correct path. The experiment should be repeated until Victor either discovers Peggy is a liar because she returned down the wrong path, or until he is sufficiently satisfied that she does indeed know the word.

This is a zero knowledge protocol because it satisfies each of the three requirements. It satisfies completeness because if Peggy knows the word she will be able to convince Victor. It is sound because if Peggy does not know the word, she will not be able to convince Victor unless she was very lucky. Finally it is zero knowledge because if Victor follows the protocol he will not be able to learn anything besides whether or not Peggy knows the word.

2.1.2 Hamiltonian Cycles

A more practical example is proving that one knows a Hamiltonian cycle for a graph, without revealing what the cycle is. Before going into the example we first need some graph theory background. A *cycle* is a sequence of vertices, two consecutive vertices in the sequence are adjacent (connected) to each other in the graph, which starts and ends at the same vertex. A *Hamiltonian path*, is a sequence of vertices in which each vertex in the graph is listed exactly once. Finally, a *Hamiltonian cycle* is a Hamiltonian path which is also a cycle. In other words it is a sequence of vertices which begins and ends with the same vertex, and each vertex in the graph is listed exactly once (aside from the first/last vertex) [8].

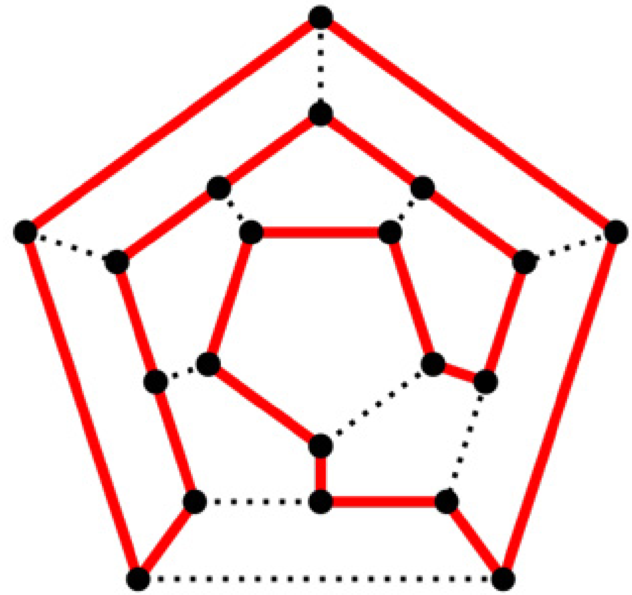

Figure 1: An example of a Hamiltonian cycle. The solid line marks the path. Taken from [8].

For a large enough graph, finding a Hamiltonian cycle is infeasible. In fact this problem, is *NP-complete*. NP-complete problems have the property that any known solution can be efficiently verified, however there is no known efficient way to find said solution. The time required to solve an NP-complete problem increases rapidly as the size of the problem grows. Using current computing power, even moderately sized problems can take billions of years to solve.

Another important definition is that of graph isomorphism. An isomorphism, $f : V(G) \to V(H)$, of graphs G and H is a bijection between the vertex sets of G and H such that any two vertices u and v of G are adjacent in G if and only if $f(u)$ and $f(v)$ are adjacent in H.

Now that we have defined a Hamiltonian cycle, we can set up the example. Here the prover, P, knows a Hamiltonian Cycle for a graph, G. The verifier, V, has knowledge of G but not the cycle. For P to show V that they know the cycle, they must perform several rounds of the following protocol.

- At the beginning of each round, P constructs H, graph which is isomorphic to G. It is simple to translate a Hamiltonian cycle between two isomorphic graphs, so since P knows a Hamiltonian cycle for G they must know one for H as well.
- P commits to H, either using a cryptographic commitment scheme or some other method. Doing this means that P cannot change H, and at the same time V has no knowledge of H.
- V then randomly asks P to do one of two things. Ei-

ther show the isomorphism between H and G, or show a Hamiltonian cycle in H.

- If P was asked to show that the two graphs are isomorphic, they start by revealing H to V. They also provide the vertex translations which map G to H. V can then verify that the two graphs are isomorphic.
- If P was asked to show a Hamiltonian cycle in H, they first translate the cycle from G onto H. They then reveal to V the edges of H which are a part of the Hamiltonian cycle. This is enough for V to verify that H contains a Hamiltonian cycle.

This protocol is complete because if P is an honest prover, they can easily answer either question asked by V by either providing the isomorphism which they have, or by applying the isomorphism to the cycle in G to demonstrate a Hamiltonian cycle. This protocol is sound because if P doesn't know the cycle, they can either generate a graph isomorphic to G or a Hamiltonian cycle for another graph, but they cannot do both since they doesn't know a Hamiltonian cycle for G. With a reasonable number of rounds it is unrealistic for P to fool V in this manner. This protocol is zero knowledge because in each round V will only learn either the isomorphism of H to G or a Hamiltonian cycle in H. V would need both pieces of information in order to reconstruct the Hamiltonian cycle in G. Therefore, as long as P can generate a distinct H each round, V will never discover the cycle in G.

3. COMPILERS

Fundamentally, what a compiler does is translate one language into another. For example, a C++ compiler will take a C++ program as input and will output machine code. There are many different types of compilers, single-pass compilers, multi-pass, load-and-go, debugging compilers, optimizing compilers, and many combinations of these [1].

The first compilers started to appear in the 1950s. Much of the early work dealt with translating arithmetic formulas into machine code. At the time compilers were notoriously difficult to implement, for instance it took 18 staff-years to implement the first Fortran compiler. Various languages, programming environments, and tools have been developed since then which make implementing a compiler considerably easier.

There are two parts to compilation, analysis and synthesis. Analysis breaks up the source into pieces and creates an intermediate representation, usually a syntax tree, of the program. Synthesis constructs the target program from the representation.

It is difficult to implement a zero knowledge protocol due to their subtleties. For this reason work has gone into developing zero knowledge compilers. A zero knowledge compiler is a compiler which takes a proof goal as its input language and outputs an implementation of a zero knowledge proof.

The compilers discussed in this paper take, as input, an abstract proof specification or proof-goal, written in languages designed specifically for this problem, and output an implementation of the given specification in a high-level language, usually C++ or Java.

4. ZERO KNOWLEDGE COMPILERS

Before discussing the three different zero knowledge compilers we require the necessary background. First, I will go over some mathematical concepts used in the proofs and compilers. After that, I will talk about the common notation used to describe zero knowledge proofs. Once familiar with that we can then begin discussing the three compilers.

4.1 Background and Notation

A *group*, in a mathematical sense, is a set, G paired with an operation, $\odot$, which combines any two elements (of the set) to form another element. A group is denoted by: $(G, \odot)$. In order for a set and operation to be a group it must meet four requirements. The set must be closed under that operation, in other words for all a, b in $G, a \odot b$ must also be in G. The operation must be associative, so for all a, b in $G, (a \odot b) \odot c = a \odot (b \odot c)$. There must be an identity element, e in G such that for every element a in G the equation $e \odot a = a \odot e = a$ is true. Finally, there must be an inverse element, so for each a in G, there exists an element b in G such that $a \odot b = b \odot a = e$. An example of a group is the integers with addition, denoted $(\mathbb{Z}, +)$.

A *preimage*, or *inverse image* of a function, $f : A \to B$, is the set of all elements a in A such that $f(a)$ is in B. For example, if $f(x) = x^2$ then the preimage of $\{4\}$ would be $\{-2, 2\}$ because those are all the elements which equal 4 after the function is applied to them.

A mapping $\phi : G \to H$ from an additive group $(G, +)$ into a multiplicative group $(H, \cdot)$ is called a *homomorphism* if and only if for all a, b in G the following equation holds: $\phi(a + b) = \phi(a) \cdot \phi(b)$ [3].

We will use notation described in [3] to denote zero knowledge proofs. An example of this notation is as follows:

$$\text{ZKP}[(\omega_1, \omega_2) : x_1 = \phi_1(\omega_1) \wedge x_2 = \phi_2(\omega_2) \wedge \omega_1 = a\omega_2]$$

What this means is *"proof of knowledge of w_1, w_2 such that $x_1 = \phi_1(\omega_1), x_2 = \phi_2(\omega_2)$ and $\omega_1 = a\omega_2$*. The common convention is that knowledge of variables denoted by Greek letters has to be proven, whereas knowledge of all other variables is assumed to be known by both the prover and the verifier. Another thing to note is that this is the notation for a *proof-goal*, not a protocol. A proof-goal describes what has to be proven, and there may be several different protocols for the same proof-goal.

4.2 Sigma-Protocols

Bangerter et al. present in [3] a language and compiler which generates sound and efficient zero knowledge proofs of knowledge based on Σ-Protocols.

Σ-Protocols are the basis of essentially all efficient zero knowledge proofs of knowledge used in practice today. Σ-Protocols are a class of three-move protocols, meaning three messages are exchanged between the prover and the verifier. First the prover, P, sends a *commitment* t to the verifier, V. V then responds with a random *challenge* c from a predefined set of challenges C. P computes a *response* s and sends it to V who then decides whether to accept or reject the proof.

Bangerter et al's compiler can handle the class of proof goals consisting of all expressions of the forms:

$$\text{ZKP}[(\omega_1, ..., \omega_m) : \bigvee \bigwedge y_i = \phi_i(\omega_i)]$$

or

$$\text{ZKP}[(\omega_1,...,\omega_m) : \bigwedge y_i = \phi_i(\omega_1,...,\omega_m) \wedge HLR(\omega_1,...,\omega_m)]$$

In the second equation, $HLR(\omega_1,...,\omega_m)$ denotes a system of homogeneous linear relations among the preimages.

Some remarks about the proof goals: The first equation can be expressed as an arbitrary monotone boolean formula, in other words a boolean formula with an arbitrarily number of $\wedge$ and $\vee$ symbols and has predicates of the form $y_j = \phi_j(\omega_j)$. Also, in both of the above equations linear relations can be proven *implicitly*: as an example, we can see that $\text{ZKP}[(\omega_1,\omega_2) : y = \phi(\omega_1,\omega_2 \wedge \omega_1 = 2\omega_2]$ is equivalent to $\text{ZKP}[(\omega) : y = \phi(2\omega,\omega)]$ by setting $\omega := \omega_2$.

The input language of this compiler requires `Declarations` of any algebraic objects involved (such as: groups, elements, homomorphims, and constants), `Assignments` from group elements to the group they live in, and `Definitions` of homomorphims. Once all of these have been set up, the protocol to be generated is specified in the `SpecifiyProtocol [...]` block.

The compiler outputs Java code for the Σ-Protocol, which can then be used in other applications. Alternatively the compiler can output LaTeX documentation of the protocol if told to do so.

4.3 ZKCrypt

Almeida et al. present, in [2], ZKCrypt, an optimizing cryptographic compiler. Similar to the above language, ZKCrypt is also based on Σ-Protocols. Using recent developments, ZKCrypt can achieve "an unprecedented level of confidence among cryptographic compilers" [2]. Specifically these developments are: *verified compilation*, where the correctness of a compiler is proved once-and-for-all, and *verifying compilation*, where the correctness of a compiler is checked on each run. ZKCrypt uses these techniques by implementing two separate compilers, one of which is a verified compiler and the other a verifying compiler. The verified compiler generates a reference implementation. The verifying compiler outputs an optimized implementation which is provably equivalent to the reference implementation.

ZKCrypt has four main parts to its compilation process. They are: resolution, verified compilation, implementation, and generation. The first phase, resolution, takes a description of a proof goal, G, as input. This description is written in the the standard notation for zero knowledge proofs. G is converted into a resolved goal G_{res}, in which high-level range restrictions are converted into proofs of knowledge of preimages under homomorphisms. The next phase, verified compilation, takes G_{res} and outputs I_{ref}, a reference implementation in the language of CertiCrypt, which is a toolset used in the construction and verification of cryptographic proofs. At this point a once-and-for-all proof of correctness is done to guarantee that I_{ref} satisfies the desired security properties. The implementation phase also takes G_{res} as input however it outputs I_{opt}, an optimized implementation. An equivalence checker is used to prove that I_{ref} and I_{opt} are semantically equivalent. In the final phase, generation, the optimized implementation is converted into C and Java implementations of the protocol.

4.4 ZKPDL

Meiklejohn et al. provide a language called the Zero-Knowledge Proof Description Language (ZKPDL) [4]. This language makes it much easier for both programmers and cryptographers to implement protocols. The authors aim to enable secure, anonymous electronic cash (e-cash) in network applications.

This language provides two main benefits. Firstly, the programmer no longer has to worry about implementing cryptographic primitives, efficient mathematical operations, or generating and processing messages. ZKPDL allows the user to specify the protocol similarly to how it would be specified in a theoretical description. Secondly, the library makes performance optimizations based on an analysis of the protocol description.

Similarly to the language above, ZKPDL makes use of Σ-Protocols. However, ZKPDL doesn't implement them directly. Instead, they use the Fiat-Shamir heuristic, which transforms Σ-protocols into non-interactive zero-knowledge proofs.

The authors also provide an interpreter for ZKPDL, implemented in C++, which preforms one of two actions depending on the role of the user. On the prover side it outputs a zero knowledge proof. On the verifier side it takes a proof and verifies its correctness. Regardless of the role of the user, the program given to the interpreter is the same. The interpreter also performs a number of optimizations including precomputations, caching, and translations to prevent redundant proofs.

Two types of variables can be declared in this language: group objects and numerical objects. Group generators can also be declared but this is optional. Numerical objects can either be declared in a list of variables or by having their type specified by the user. Valid types are: `element`, `exponent`, and `integer`.

A program written in this language is split into two blocks: a computation block, and a proof block. Both blocks are optional, if the user is only interested in the computation they can just write that. Alternatively, if the user has all the computations done they can just write the proof block.

The computation block can be further split into two blocks: the `given` block and the `compute` block. In the `given` block the parameters are specified as well as any values which are necessary for the computation that the user has already computed. The `compute` block carries out the given computations. There are two types of computations: picking a random value, and defining a value by setting it equal to the right-hand side of an equation.

The proof block is made up of three blocks: the `given` block, the `prove knowledge of` block, and the `such that` block. In the `given` block the proof parameters are specified as well as any inputs known publicly to both the prover and the verifier. The inputs known privately to the prover are specified in the `prove knowledge of block`. In the `such that` block the relations between all the values are specified. The zero-knowledge proof will be a proof that all these relations are satisfied.

5. APPLICATIONS

In general, zero knowledge protocols have many applications. Authentication systems, electronic voting, electronic ticketing, Direct Anonymous Attestation (DAA), and Off-the-Record messaging [2, 4] are just a few examples. The applications that will be focused on in this paper are electronic cash, and deniable authentication.

5.1 Electronic Cash

Electronic Cash, or e-cash, is an electronic currency. E-cash maintains the buyer's anonymity, unlike a debit or credit card that is used to purchase something electronically. Bitcoins are a recent example of an e-cash system.

Okamoto and Ohta describe the ideal electronic cash system in [6]. The ideals are as follows:

1. *Independence*: The security of electronic cash cannot depend on any physical condition. Then the cash can be transferred through networks.
2. *Security*: The ability to copy (reuse) and forge the cash must be prevented.
3. *Privacy (Untraceability)*: The privacy of the user should be protected. That is, the relationship between the user and their purchases must be untraceable by anyone.
4. *Off-line payment*: When a user pays the electronic cash to a shop, the procedure between the user and the shop should be executed in an off-line manner. That is, the shop does not need to be linked to the host in user's payment procedure.
5. *Transferability*: The cash can be transferred to other users.
6. *Dividability*: One issued piece of cash worth value C (dollars) can be subdivided into many pieces such that each subdivided piece is worth any desired value less than C and the total value of all the pieces is equivalent to C.

Almeida et al. describe briefly in [2] how ZKCrypt can be used to generate a proof for proving the identity of the user when withdrawing money from a bank account. The user has to prove they have a secret key in order to successfully withdraw money.

The authors state the proof goal as:

$$ZPK\left[(u_1, u_2) : I = g_1^{u_1} g_2^{u_2}\right].$$

In this goal, $I, g_1, g_2 \in \mathbb{Z}_p^*$ such that ord $g_1 =$ ord $g_2 = q$, where $q|(p-1)$ and $p, q \in \mathbb{P}$. The secrets u_1, u_2 are elements of $\mathbb{Z}_q$. A single instance of the Σ^{Φ}-protocol is enough to realize this goal.

Meiklejohn et al. also give this example, a user proving their identity to the bank, implemented in ZKDPL. The program for this looks like:

```
proof:
  given:
   group: cashGroup = <f, g, h, h1, h2>
   elements in cashGroup: A, pk_u
     commitment to sk_u: A = g^sk_u * h^r_u
  prove knowledge of:
   exponents in cashGroup: sk_u, r_u
  such that:
   pk_u = g^sk_u
   A = g^sk_u * h^r_u
```

When the bank has verified this proof, the bank and the user will run a protocol which defines a wallet which contains W coins, where W is a system-wide public parameter. When a user spends a coin, it is split up into two parts: an endorsed part and an unendorsed part. Separately the two parts are worthless, but together the coin becomes valid. First the unendorsed part is sent to the vendor who proves its validity. The vendor then sends what the buyer has purchased. The buyer sends the endorsed portion of the coin to the vendor upon receiving their product.

6. CONCLUSION

Zero knowledge protocols are becoming more and more important in today's society. Because they can be increadably complex and take a while to implement, it is important that there are efficient and secure ways of implementing them. In this paper we discussed three zero knowledge compilers and thier associated input languages. Each of these compilers hopes to aid in the implementation of zero knowledge protocols by offering a way for developers to easily generate an implementation of a given proof goal.

6.1 Current State and Future Work

Currently, all three of the compilers presented in this paper have been implemented in some form. The compiler outlined in [3] has native support for two groups and allows users to define their own groups. Future features of the compiler includes supporting efficient proofs in hidden-order groups, as well as the automatic transformation of the generated Σ-protocols into non-interactive zero knowledge proofs.

ZKCrypt has been implemented and applied to several cryptographic problems, including electronic cash and deniable authentication. Future work for the ZKCrypt compiler includes verifying the last stage of the compiler chain, code generation. With this verified, the whole compilation process will be verified correct.

ZKPDL has also been implemented and has been applied to problems such as electronic cash and verifiable encryption. Future work being considered for ZKPDL includes adding more cryptographic primitives, such as: encryption, signatures, and hash functions. Another interesting possibility would be the analysis of ZKDPL programs by providing automatic verification of protocols and the ability to identify security errors. Work is also being done to increase performance on multicore architectures by analyzing dependencies among expressions evaluated by the interpreter.

6.2 Acknowledgments

// TODO

7. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] J. B. Almeida, M. Barbosa, E. Bangerter, G. Barthe, S. Krenn, and S. Z. Beguelin. Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 488–500, New York, NY, USA, 2012. ACM.

[3] E. Bangerter, T. Briner, W. Hencecka, S. Krenn, S. Ahmad-Reza, and T. Schneider. Automatic generation of sigma-protocols. In *Proceedings of the 6th European conference on Public key infrastructures, services and applications*, EuroPKI'09, pages 67–82, Berlin, Germany, 2009. Springer-Verlag.

[4] S. Meiklejohn, C. C. Erway, A. Kupcu, T. Hinkle, and A. Lysyanskaya. Zkpdl: a language-based system for efficient zero-knowledge proofs and electronic cash. In *USENIX Security'10 Proceedings of the 19th USENIX conference on Security*, Security '10, Berkeley, CA, USA, 2010. USENIX Association.

[5] A. Mohr. A survey of zero-knowledge proofs with applications to cryptography.

[6] T. Okamoto and K. Ohta. Universal electronic cash. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, pages 324–337, London, UK, UK, 1992. Springer-Verlag.

[7] J.-J. Quisquater, L. Guillou, M. Annick, and T. Berson. How to explain zero-knowledge protocols to your children. In *Proceedings on Advances in cryptology*, CRYPTO '89, pages 628–631, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[8] Wikipedia. Hamiltonian path — wikipedia, the free encyclopedia, 2013. [Online; accessed 14-November-2013].

[9] Wikipedia. Zero-knowledge proof — wikipedia, the free encyclopedia, 2013. [Online; accessed 1-November-2013].