Zero Knowledge Compilers

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

ABSTRACT

<Insert Abstract Here>

General Terms

Need to figure this out yet

Keywords

Zero Knowledge Protocols, Compilers

1. INTRODUCTION

I will focus on Zero-Knowledge Compilers which are compilers that automatically generate Zero-Knowledge proofs. This is how I plan to use the following sources:

- I expect [2, 3, 5] to be my core sources, depending on how relevent [5] turns out to be I'll replace it with a better source.
- I will use [4, 6, 7] for background information and examples of Zero-Knowledge Protocols.
- I will need to find some papers for background information on compilers.

As stated above I need to find some sources about compilers. I probably will need to find more papers dealing with ZK-Compilers as well.

1.1 Key Points

What main problems(s) or questions(s) does the research address?

The main problem that the research address is how to create reliable zero knowledge protocols. They can be difficult to define and even harder to verify. Zero knowledge compilers help because they can efficiently generate zero knowledge protocols, and because of how they are constructed the user can trust that they will work.

What are the key contributions of each of your main sources?

Source [2] provides a great deal of information about their zero knowledge compiler, ZKCrypt. They go into detail about zero knowledge protocols, how their compiler produces them, and they give a proof verifying that their protocols are valid. They also talk about a few applications of their compiler.

Source [5] talks in depth about ZKPDL, which is a language they created for writing zero knowledge protocols. They also created an interpreter for this language, which performs optimizations to lower computational and space overhead. This paper also provides an example dealing with electronic cash.

Source [3] uses Σ-Protocols in a compiler to automatically generate sound and efficient zero knowledge proofs of knowledge. The compiler automatically generates the implementation of the protocol in Java, or it can output a description of the protocol in LaTeX.

How are the main sources related to each other?

The main sources all use compilers to generate zero knowledge protocols, but the ways they are implemented are all different so there is some room for comparison. All the compilers are also based off Σ-Protocols, or variations of Σ- Protocols.

What is the state of the research?

The current state is that the compilers have been implemented and tested. They all provided enough data to back up their research. Most of the work they are doing now will extend the applications of their compilers to support other proof types.

What background material will you need to present in order for your audience to understand the research?

I will need to provide background information on zero knowledge protocols and compilers. It's probably more important that I focus on zero knowledge protocols and only give basic compiler background.

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* [8]. Formally, a zero knowledge proof is a type of interactive proof.

An interactive proof system for a set S is an interaction between a *verifier* executing a probabilistic polynomial-time

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

strategy and a *prover* which executes a computationally unbound strategy satisfying:

- *Completeness*: For every $x \in S$, the verifier always accepts after interacting with the prover on common input x.
- *Soundness*: For every $x \notin S$, the verifier rejects with probability at least $\frac{1}{p(|x|)}$.

To summarize: if an honest verifier is always convinced after interacting with an honest prover, then the proof is complete. A proof is sound if a cheating prover can only convince an honest verifier with some small probability [6]. For an interactive proof to be a zero knowledge proof it must also satisfy the condition of *zero knowledge*.

A proof is zero knowledge on (inputs from) the set S if, for every feasible strategy B^*, i.e. a strategy that is both sound and complete, there exists a feasible computation C^* so that the following two probability ensembles are computationally indistinguishable:

- the output of B^* after interacting with A on common input $x \in S$
- the output of C^* on input $x \in S$

In other words, any information, in this case B^*, that can be learned by interacting with A is the same as the information that can be learned without interacting with A [6].

2.1 Examples

Below are two examples of zero knowledge protocols. The first example is easy to follow and just highlights how a zero knowledge protocol functions. The second example is more in depth and shows how an application of zero knowledge protocols.

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 [6] 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.

2.2 Notation

3. COMPILERS

There are many different types of compilers, single-pass compilers, multi-pass, load-and-go, debugging compilers, optimizing compilers, and many combinations of these. Different compilers do different things, but at their core all compilers must perform one function. Simply put, they must take a program as an input and output an equivalent program in a different language. [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.

4. DEFINITIONS/TERMS

I might need a small section here detailing some crypto/number theory concepts that will be needed below.

5. ZERO KNOWLEDGE COMPILERS

Due to the subtleties of implementing a zero knowledge proof they can be difficult to get correct. 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.

5.1 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:

$$\text{ZKP}[(w_1, ..., w_m) : \bigvee \bigwedge y_i = \phi_i(w_i)] \quad (1)$$

or

$$\text{ZKP}[(w_1, ..., w_m) : \bigwedge y_i = \phi_i(w_1, ..., w_m) \wedge HLR(w_1, ..., w_m)] \quad (2)$$

In (2), $HLR(w_1, ..., w_m)$ denotes a system of homogeneous linear relations among the preimages.

Some remarks about the proof goals: Equation (1) 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 $y_j = \phi_j(w_j)$. Also, in both of the above equations linear relations can be proven *implicitly*: as an example, we can see that $\text{ZKP}[(w_1, w_2) : y = \phi(w_1, w_2 \wedge w_1 = 2w_2]$ is equivalent to $\text{ZKP}[(w) : y = \phi(2w, w)]$ by setting $w := w_2$.

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 LaTeXdocumentation of the protocol if told to do so.

5.2 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.

5.3 ZKPDL

Meiklejohn et al. provide a language called the Zero-Knowledge Proof Description Language (ZKPDL) [5]. 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 speci-

fied. The zero-knowledge proof will be a proof that all these relations are satisfied.

6. 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, 5] are just a few examples. The applications that will be focused on in this paper are electronic cash, and deniable authentication.

6.1 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 [**?**]. 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.

6.2 Deniable Authentication

7. CONCLUSION

This is where I'll neatly wrap everything up.

8. 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] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Advances in Cryptology Proceedings*, CRYPTO' 89, pages 218–229, New York, NY, USA, 1987. ACM.

[5] 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.

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

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