University of Tartu
Faculty of Mathematics and Computer Science
Institute of Computer Science

Margus Niitsoo

Can we Construct Unbounded Time-Stamping Schemes from Collision-Free Hash Functions?

Master's thesis

Supervisor: Ahto Buldas, PhD

Tartu 2008

Contents

Introduction

Suppose you are an inventor and you want to be able to prove you had a certain ingenious idea before anyone else. However, you do not want to reveal what the idea is. You thus just want your documents to be tied to the time they were created, possibly without revealing the contents of the documents themselves. This is the problem of time-stamping. Simple forms of time-stamping have been around for hundreds of years via the notary and patent offices and even the postal service has been used to this effect, but in the current age of electronic communication such archaic methods may often be too slow. Cryptographic time-stamping schemes can be used to remedy that situation.

However, constructing a scheme that is reliable but also secure against forgeries is by no means a trivial matter. Many of the models originally discussed have turned out to be insecure under the original assumptions. This does not mean they are insecure but means that their security cannot be proven in the way originally thought plausible and that either stronger or different assumptions are needed. One of the best known examples is the unbounded hash tree based approach of Harber and Stornetta that was later analyzed by Saarepera and Buldas.

This thesis concentrates on the impossibility of constructing a secure hash function for the unbounded time-stamping scheme Saarepera and Buldas proposed from collision-resistant hash functions. Impossibility results of this type are usually proved via so called oracle separation methods by showing that given use of a certain oracle, one of the primitives exists and the other doesn't. We mainly concern ourselves with studying the properties of one candidate for such an oracle

– one that constructs a large hash tree and then uses that to give time-stamping certificates. We study the possibilities of exploiting this oracle to find collisions for hash functions and show that constructing an adversary that could actually do that is quite complicated.

The first chapter gives a gentle introduction into the unbounded time stamping scheme described by Harber and Stornetta and shows the origins of the problem. The second chapter is mainly dedicated to cryptographic reductions and methods of proving that none can exist between two primitives. The third chapter is composed of original results that study the properties of the proposed hash tree oracle and show that the simplest possible approach the adversary could take can be foiled by a cleverly constructed oracle. The fourth chapter discusses some other approaches that can be ruled out in a similar manner and then goes on to discuss other possible constructions for separation oracles.

1. Unbounded Timestamping

1.1 Timestamping Introduction

Suppose you are an inventor and you just had a brilliant idea. You want to protect yourself against someone later claiming to have had that same idea, but earlier. If he honestly claims so, there is relatively little you can and should be able to do. To avoid a dishonest claim of this type, it would be sufficient if you could securely tie your idea to the date and possibly time at which you discovered it. This is the problem of timestamping.

One simple model would be to have a trusted central authority that recieves the idea, appends a timestamp to it in a standard way and then signs it. Assuming that the signature scheme is secure and the central authority is always trustworthy, this is a good model and it has been used in paperwork for hundreds of years – notary and patent offices essentially fill this role.

Assume however that the inventor does not want to trust a central authority with his idea. Over the years, many rather ingenious things have been done to the effect of timestamping. The most common of them involved paranoid inventors sending the documents describing the invention to themselves in a sealed envelope. The postal timestamp could later be used in court, assuming the envelope was left sealed until that time.

The digital version of that scheme would be to send a hash value computed using a publicly known hash function of your invention to the central authority instead of the plaintext document. Assuming that the hash function is hard to reverse, this trusts relatively little information about your actual work into the hands of the authority. When the timestamp needs to be verified, the document can be presented and people can check that the signature is indeed given to the hash value corresponding to it.

This scheme is not without its problems. The main one is that it is rather easy for the authority to issue backdated timestamps, so it may be possible for someone to get a timestamp to their document that claims he had the results earlier, when he didn't. We would like to make it hard for even the central authority itself to forge timestamps. This can indeed be achieved and many different models have been built that do so.

This thesis works with the model proposed by Harber and Stornetta in [6]. The the security of this model was extensively scrutinized by Buldas and Saarepera in [4] and it was shown that collision-resistance, one of the most common properties expected from a hash function does not imply that it is secure for that scheme. The main aim of this thesis is to explore the possibility that no constructions of secure hash functions for this scheme could be made from collision resistant functions. Before going into the details of security, we first explain the scheme itself.

1.2 The Scheme of Harber and Stornetta

1.2.1 Parties Involved

The scheme of Harber and Stornetta involves three parties: a Client C, a Server S and a repository $\mathcal{R}$ and gives two procedures – one for creating a timestamp and one for verifying it. It is assumed that $\mathcal{R}$ is write-only so once something is commited to it, it cannot be changed. This can be accomplished in practice by publishing the value in a widely available medium so many different and unaffiliated parties can mirror it (a quote from Linus Torvalds: "Only wimps use tape backup: real men just upload their important stuff on ftp, and let the rest of the world mirror it"). The server is in the role of a "trusted authority" but is severely more restricted than in the naive notary or postal stamp models described in the previous section.

The scheme uses two hash functions $h_c\colon \{0,1\}^* \to \{0,1\}^k$ (that is, a function from a bitstring of any length to one of length k) and $h_s\colon \{0,1\}^{2k} \to \{0,1\}^k$. The function h_c is the function used by the clients to get hash values from original documents. The other, h_s is the server side function which is used for computing the published hash value.

1.2.2 Hash Circuits

In this work we are mainly concerned with the properties of the server-side function h_s. We note that it is defined to be from $2k$ length bitstrings to k length bit stings. We can therefore model h_s as a function with two inputs of length k and with one output of the same length. This allows us to write $h_s(x_1, x_2) = y$ where $x_1, x_2, y \in \{0,1\}^k$. We can also model them as circuit elements (or "gates") where we assume that each "wire" carries k bits simultaneously (see fig. 1.1).

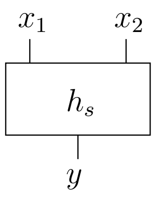

Figure 1.1: Diagram for $h_s(x_1, x_2) = y$.

Since the output is of the same length as the inputs, we can use the output of one of these elements as an input for another one. This allows us to build trees of the h_s gate, taking many different inputs $x_1, \ldots, x_m$ but giving only one single output r_t. There are many ways of constructing a tree given a fixed number of inputs (see fig. 1.2 for two example trees for four inputs - the first depicts $h_s(h_s(x_1, x_2), h_s(x_3, x_4))$ and the second is for $h_s(h_s(h_s(x_2, x_4), x_1), x_3))$. From now on we call such trees composed of hash functions either *hash trees*, *hash circuits* or *Merkle trees* in honour of Ralph Merkle who invented them in 1980 [10]. We call the value r returned at the bottom of the tree its *root value.*

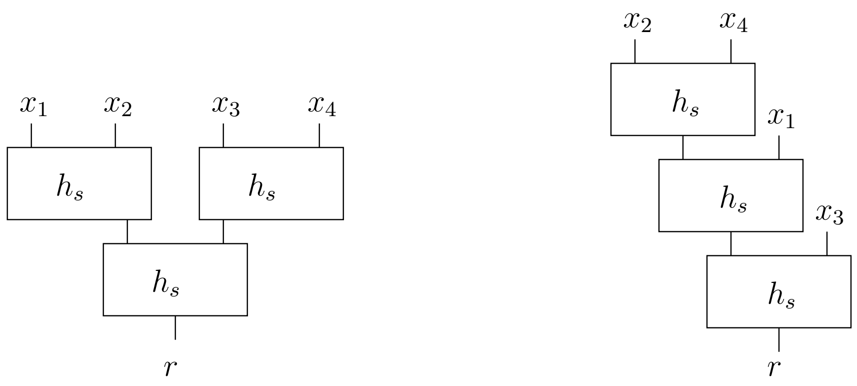

Figure 1.2: Examples of hash trees with 4 inputs.

1.2.3 Time-stamping Procedure

Time-stamping procedure is divided into rounds of equal duration. During each round, the server S waits for hash values of length k assumably formed by h_c to be sent to it by the clients C. At the end of the round t, it takes all the values $x_1, \ldots, x_m \in \{0,1\}^k$ sent to it by the clients and then builds a hash circuit out of them. It then calculates the root value r_t. How the binary tree is constructed is chosen by the server S and is by the initial scheme not restricted in any way, assuming that all inputs sent to S are used in it. For example, both trees from fig. 1.2 could be used in case of four inputs. This is why the scheme is called unbounded.

The server then publishes the value r_t into the repository $\mathcal{R}$ and starts sending out certificates to the clients. The certificate itself is an ordered 4-tuple $c = (x, t, n, z)$ where x is the value being certified, t is the number of the round that just ended, $n = n_1 n_2 \ldots n_l$, $n_i \in \{0,1\}$ describes the path from x down to the root and $z = (z_1, \ldots, z_l) \in (\{0,1\}^k)^l$ gives the information to verify that path.

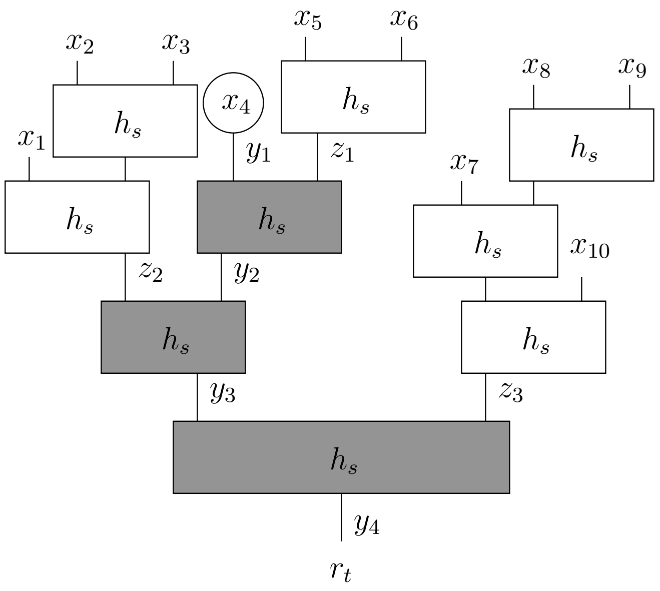

Figure 1.3: A larger tree with a path marked from x_4.

We give a small example by describing the certificate of x_4 in fig. 1.3. The sequence n encodes the structure of the path starting from the original value x_4 downwards while the sequence z gives the other inputs used alongside x_4 and values calculated from that. As x_4 is the left input for the first box, $n_1 = 1$ and we take z_1 to be the other input into that box (which in this case is $h_s(x_5, x_6)$). We now move to the second box on the path and see that the output of the previous box is now the right input. We thus set $n_2 = 0$ to signify that and then take z_2 to be the second input into this box (which in this case is $h_s(x_1, h_s(x_2, x_3))$). The third and final element on the path has the second one as the left input again so we set $n_3 = 1$ and take z_3 to be the other input again. Since there are no more boxes, the certificate for x_4 for the tree given in fig. 1.3 is $c = (x_4, t, 101, (z_1, z_2, z_3))$.

1.2.4 The Verification Procedure

The verification procedure can be carried out by C based on his original document D and the certificate $c = (x, t, n, z)$ issued for it. The first step is to check that the x in the certificate indeed matches the hash value $h_c(D)$ of the original document. The calculation then proceeds by defining $y_1 := x$ and then inductively calculating the sequence $y_2, \ldots, y_{l+1}$ based on the formula

$$y_{i+1} := \begin{cases} h_s(z_i, y_i) & n_i = 0 \\ h_s(y_i, z_i) & n_i = 1 \end{cases} . \tag{1.1}$$

Once it has the value y_{l+1} it queries the repository $\mathcal{R}$ for the value r_t and checks whether $r_t = y_{l+1}$. We refer the reader back to fig. 1.3 for an illustration of z and y values in the circuit.

For notational convenience we define the verifier function $V(x, n, z) := y_{l+1}$. Then the last paragraph describes how to check whether $V(x, n, z) = r_t$.

Essentially, a certificate for x thus consists of the path that leads from x down to the root, where n specifies which direction the path turns to and z gives all the other values used in that path alongside y_i.

1.3 Security of the scheme

1.3.1 Introduction to Cryptology

Cryptology is the science of secure communication in the widest sense. It has roots dating back to antiquity but arose in its modern form only in the last century. The main goal of modern cryptology is to construct communication schemes that are secure relative to certain possible attacks.

We bring a small example from the time-stamping scenario described above. Suppose there is a malicious adversary that intercepts the value of $h_c(D)$ as it is transmitted from client C to server S. Suppose that this adversary is somehow capable of deducing vital details about D from that value. This would be considered an attack. If C wants to be secure against this type of attack, he has two options – use a secure channel that cannot be eavesdropped or use a hash function h_c that does not reveal any useful information about D from $h_c(D)$. The second would be considered a security property of that h_c.

Cryptology research can be divided into two broad categories. The first one is trying to construct certain secure primitives such as functions that are hard to reverse like we would have needed in the previous example. The second tries to use these primitives in more complicated schemes such as the time-stamping scheme described in the previous section and to prove the security of these schemes on the assumption of the security of the primitives.

The main problem of applied cryptology is that practically no provably secure primitives actually exist. The security of most of them rests on different types of assumptions (large integers being hard to factor, discrete logarithm problem being hard for certain groups, $\mathrm{P} \neq \mathrm{NP}$). Also, given infinite amount of time, most primitives can easily be shown to be breakable. Therefore the notion of security is usually defined in terms of a bounded time adversary being able to gain only a mariginal advantage. We now present a concrete example of that paradigm which we can use to explain what we mean exactly.

1.3.2 Understanding Cryptographic Security

We first define the notion of a collision that is central in the presentation of this thesis:

Definition 1.3.1. We say that the pair $x_1, x_2 \in \{0,1\}^n$ form a *collision* for $h : \{0,1\}^n \to \{0,1\}^m$ if $x_1 \neq x_2$ and $h(x_1) = h(x_2)$.

We now bring a textbook security property definition and then try to describe what is meant by it in simpler terms.

Definition 1.3.2. We say that a family χ of hash functions $h\colon \{0,1\}^n \to \{0,1\}^m$ $(n > m)$ is (t, ϵ)-*collision resistant* if for any t-time adversary A we have

$$\Pr[h \leftarrow \chi, (m_0, m_1) \leftarrow A(h)\colon m_0 \neq m_1, h(m_0) = h(m_1)] < \epsilon \ . \qquad (1.2)$$

We first start by specifying what is a t-time adversary. By an adversary we usually mean an algorithm (a computer program) that is given certain input and produces a certain output corresponding to that input which has some sort of undesirable properties. In our example, the adversary A is given as input the description of h and is expected to output a pair of values (m_0, m_1) (which is described in the definition by $(m_0, m_1) \leftarrow A(h)$). We call an adversary program t-time adversary if it makes at most t steps in its execution before producing an output in some computational model. What that model is, does not usually concern us, so it can be a Turing machine, a random access machine or any other reasonable model of that type. It could also be a human being who is given exact instructions on what to do (so he cannot use his creativity) and has at most t minutes of time before he is required to produce an output. By the Church-Turing thesis all such models are equal and although some are faster and some are slower, we are usually not concerned about the specific implementation. Therefore one model is usually just fixed, Turing machine being the most common choice for that.

However, the adversary does not even have to be wholy deterministic and in fact is often considered to have access to an infinite supply of random cointoss results on which to base his random descisions. In essence a t-time adversary is thus an adversary (a computer program or a non-creative human being) that has only a limited amount of time t before an output is expected from it and that works according to fixed rules, but may use randomness in his descisions. Due to the fact that the algorithm might be randomized, its output is not one specific value but rather chosen from a certain distribution based on the randomness distribution given to it.

The equation 1.2 can be stated in words: "The probability that after randomly choosing a h from χ the two values m_1 and m_2 returned by the adversary A with input h are different but $h(m_1) = h(m_2)$ is less than ϵ". We call such a pair of (m_1, m_2) a collision and if the adversary has a low chance of finding one for this family of hash functions χ we call the family collision resistant.

There are several reasons that security is defined like it is:
Firstly, we define collision resistance on a family of hash functions because for every single function there exists an adversary that finds a collision: since there are more possible input values (2^n) than output values (2^m), there has to be at least one such pair (m_1, m_2) and we can take an adversary that doesn't even look at the input and just blindly returns this pair. It is guaranteed to find a collision for the fixed h (although it fails to find a collision for most of the others).
Secondly, we bound the time the adversary is allowed to work because if we did not, it could just try all the possible pairs until one fit and then return it. This would, however, take at least roughly $2^n \cdot 2^n = 2^{2n}$ steps so if we choose t to be a lot smaller (say n or n^2), that tactic would not lead to a good chance of finding a collision.
Thirdly, we use ϵ instead of 0 because the latter is infeasible – since we know that given infinite time, we can always find a collision, it is only rational (and can in fact be proven) that the adversary does gain a small advantage even if it only works for a short time (and as the time bound increases, so does the advantage).

This is the basic model the security properties of the primitives are defined in - we have a time-bounded adversary that is allowed to gain a small advantage (just as long as it is small). The security of the schemes is usually defined for roughly the same model. We now go on to investigate the security of the Harber-Stornetta scheme.

1.3.3 The Actual Security of Harber-Stornetta Scheme

The original authors considered security against the attack where the bounded time adversary is allowed to commit hash values $x_1, \ldots, x_n$ to timestamping, receives their certificates from the server and the root value from the repository. He is then expected to produce x that is different from $x_1, \ldots, x_n$ and a certificate for it that would be valid for the original period.

This is a very strong security claim, which means that no value x could possibly be backdated at a later time with a reasonable probability. The original paper postulated that the scheme is secure in that respect if h_s was chosen to be collision resistant. The curious reader may start to wonder what collision resistance could possibly have to do with the security of this type of scheme against this type of attack.

The scheme was critically revised by Buldas and Saarepera in [4]. They noted that the scheme is in fact insecure against the described type of attack. Simply put, the adversary could always randomly choose $x, y \in \{0,1\}^k$, compute $w = h_s(x, y)$ and then commit w. Upon getting a valid certificate $c = (w, t, n, z)$ for w from the server it could construct a valid certificate for x by appending 1 to n and y to z so $c_x = (x, t, n||1, z||y)$ would be a valid certificate. Therefore, unless $w = x$, the scheme would be "broken". After noting that the same trick could be used for producing a certificate for y as well and that we can choose $x \neq y$ we can always backdate at least one value.

They however noted that this attack has a critical flaw - the adversary had to know what he wanted to backdate (the x and y values) when he commited his values to timestamping. It follows that this type of attack does not really jeopardize the security of this scheme in practical applications. They then gave a security condition that better describes a real-world attack scenario.

The new scenario is this: We assume that the server may coerce with the adversary and may allow it to commit a few values r_i into the repository $\mathcal{R}$. The adversary then waits for something to backdate (not knowing what it may be). Once anything of that type arrives (for instance a new invention he wants to claim patent rights to), he tries to backdate it to one of the values r_i previously commited by him. If he succeeds, the attack is considered successful.

This differs from the previous attack model mainly by that the adversary has no knowledge what he has to backdate when he is allowed to commit the original values. For this, the adversary needs to be broken up into two parts - one that finds the values to be commited and the other that tries to produce a certificate for a value fed to it for backdating. The new security property is defined in the following way:

Definition 1.3.3. We say that the Harber-Stornetta scheme is (t, ϵ)-secure relative to the distribution $\mathcal{D}$ if for any t-time adversary $A = (A_1, A_2)$

$$\Pr[(r, a) = A_1, x \leftarrow \mathcal{D}, (n, z) = A_2(x, a)\colon V(x, n, z) = r] < \epsilon \ . \tag{1.3}$$

Note that the small a is just extra information that A_1 passes to A_2 as they are still in essence one program.

Buldas and Saarepera then go on to show that the original scheme is secure in that respect if it uses a collision-free hash function assuming that the structure of the trees being constructed is restricted and verification checks whether the certificate is consistent with the tree structure. This version of timestamping is called restricted-tree timestamping and since rather good approximations to collision-free hash functions exist, a scheme implemented that way can be made secure.

1.4 Chain Resistance Property

The security of the unbounded case still remains a problem, however. Buldas and Saarepera introduce a new security property for hash functions called chain resistance for just that purpouse.

Definition 1.4.1. A hash function $h\colon \{0,1\}^{2k} \to \{0,1\}^k$ is (t, ϵ)-*chain resistant* (relative to a distribution $\mathcal{D}_k$ on $\{0,1\}^k$) if for every t-time adversary $A = (A_1, A_2)$

$$\Pr[(r, a) = A_1, x \leftarrow \mathcal{D}_k, (n, z) = A_2(x, a)\colon V(x, n, z) = r] < \epsilon \ . \tag{1.4}$$

This essentially means that for a randomly chosen $x \leftarrow \mathcal{D}_k$ it is hard to construct a chain from h that leads from x down to r. This definition is rather directly adopted from the security condition for the Harber-Stornetta scheme. As such, it is rather trivial to show that if h_s used in the scheme is (t, ϵ)-chain resistant, then the scheme itself is (t, ϵ)-secure.

The definition nearly repeats the security condition for the Harber-Stornetta scheme. However, in this case, it is not a property of a scheme but rather the property of a hash function. The main question that arises is: Do hash functions with this property actually exist and can we construct them? The simplest way of doing this would be proving that a certain already known security property such as collision resistance would automatically imply chain resistance.

Buldas and Saarepera showed that no standard reductions used in cryptography today could allow one to prove that collision resistance implies chain resistance. They conjecture that a so-called black-box construction of a chain resistant function from a collision-resistant one may also be impossible. The topic of this thesis is to examine that hypothesis.

2. Cryptographic Reductions

2.1 The Construction of Merkle and Damgård

To illustrate what a black-box reduction actually means, we begin this chapter with a theorem due to Merkle [9] and Damgård [5] who proved it independently. To illustrate the reduction better, we use a somewhat simplified form of the original construction. The reader more interested in how such constructions are used in practice should consult a good cryptographic textbook (for instance [14]).

Theorem 2.1.1. *Assume that for fixed $n, m \in \mathbb{N}$ there exists a family $\mathcal{F}_1$ of collision-resistant hash functions $f\colon \{0,1\}^{n+m} \to \{0,1\}^m$. Then for every $k \in \mathbb{N}$ there also exist a family $\mathcal{F}_2$ of hash functions $h\colon \{0,1\}^k \to \{0,1\}^m$ that is collision-resistant.*

Proof. We start by constructing the family $\mathcal{F}_2$. Let $f\colon \{0,1\}^{n+m} \to \{0,1\}^m$ be any function from $\mathcal{F}_1$ and let $s \in \{0,1\}^m$ be a randomly chosen seed. We now define a function $h_{f,s}\colon \{0,1\}^k \to \{0,1\}^m$ by showing how it works on a fixed input $x \in \{0,1\}^k$. The family $\mathcal{F}_2$ can then be defined as the set of all such functions $h_{f,s}$ where $f \in \mathcal{F}_1$ and $s \in \{0,1\}^m$.

If n does not divide k, we begin by adding zeroes to the end of x until its length is a multiple of n. We then break x into blocks of n bits so $x = x_1|x_2|\cdots|x_l$ for $l = \lceil \frac{k}{n} \rceil$. After that, we construct $y_1, \ldots, y_l \in \{0,1\}^m$ by specifying $y_1 = f(s|x_1)$ and $y_i = f(y_{i-1}|x_i)$ for $i = 2, \ldots, l$. The value y_l is then returned as the output of $h_{f,s}(x)$.

The preceeding description of computation can easily be formalized as an algorithm. It should also be easy to see that if f can be computed in t steps then h can be computed in roughly $lt + n$ steps so it remains relatively efficient. It is also crucial to note that we use f in a black-box manner – we do not know how it works, only that it does. We give it input and it gives us output, but how it computes the output does not concern us.

We now need to show that if $\mathcal{F}_1$ is collision-resistant then so is $\mathcal{F}_2$. Assume the opposite, eg. that $\mathcal{F}_1$ is indeed collision-resistant but that $\mathcal{F}_2$ is not. There then exists an adversary A that can break the collision-resistance property for the functions $h \in \mathcal{F}_2$ with more than a negligible probability. Assume A can find a collision pair (a, b) for $h_{f,s} \in \mathcal{F}_2$. Let $a = a_1|a_2|\cdots|a_l$ and $b = b_1|\cdots|b_l$ where both a_i and b_i are all blocks of length n bits where a and b are padded with zeroes if needed. Then the computation of $h(a)$ yields a sequence $a'_1, \ldots, a'_l$ and the computation of $h(b)$ gives $b'_1, \ldots, b'_l$. Since (a, b) is a collision, we have $a'_l = h(a) = h(b) = b'_l$. This implies that $f(a'_{l-1}|a_l) = f(b'_l)$. If $a'_{l-1}|a_l \neq b'_{l-1}|b_l$, this gives us a collision for f. If not, let r be the smallest such value that $a'_{r+1} = b'_{r+1}$ but $a'_r|a_{r+1} \neq b'_r|b_{r+1}$. This value has to exist because (a, b) is a collision so $a \neq b$ which implies that a_r and b_r differ at some point. It is also clear that it gives us a collision for f. Therefore, we can also break $f \in \mathcal{F}_1$ by choosing a random seed s, using A to find a collision for $h_{f,s}$ and use that to find one for f.

We note that we essentially constructed an adversary for $\mathcal{F}_1$ based on an adversary A for $\mathcal{F}_2$. The construction is efficient because we essentially follow the same steps as in the computation of h. This implies that $\mathcal{F}_1$ cannot be collision-resistant which contradicts our original assumption. Therefore, $\mathcal{F}_2$ has to be collision resistant if $\mathcal{F}_1$ is. $\square$

We note that the proof if fully constructive – we show how to construct $\mathcal{F}_2$ and then show how to break $\mathcal{F}_1$ if we know how to break $\mathcal{F}_2$. This proof technique is known as a reduction – we reduce the problem of the security of $\mathcal{F}_2$ to the security of $\mathcal{F}_1$.

2.2 Cryptographic Practice

Reductions of this type are one of the main tools in cryptography. The idea is to prove the security of a complex scheme based on the security of its constituents by showing that if there exists an adversary that breaks the complex scheme then we can construct an adversary that breaks at least one of the constituent primitives. Since the security of the primitive is taken as a premise, this gives a contradiction and thus the scheme must be secure. However, since we want the scheme to work regardless of the actual primitives being used, we cannot make any assumptions about them other than them being efficient. This gives rise to so-called black-box reductions – we are assumed to be presented with a method of implementing the original primitive (the black box) and we can use it, but we have no idea on how it is constructed.

Cryptographic reductions are similar in many ways to complexity-theoretic reductions. While cryptography studies the security of certain problems then complexity theory is more interested in how efficiently something could be computed in theory. It is clear that complexity theory plays a rather important role in cryptography as well, since the notion of being easily computable is used quite often – we want the primitives to be easily computable but the adversaries not to be so. It follows that many cryptologists today have a rather strong background in complexity theory.

Complexity-theoretic reductions are usually used to prove that one problem is no less hard to solve for a computer than the other. However, as complexity theory is more interested in general limits than on specific problems, there is a large body of theorems that show that certain types of problems cannot be reduced to certain others via certain types of reductions. The reductions were introduced into cryptology in the early eighties by Micali and it did not take long for cryptologists to start coming up with the same types of theorems for the limits of cryptographic reductions. For that, however, they first needed to formalize the notion of a cryptographic reduction. Before introducing the formalizations, we need to introduce some complexity-theoretic preliminaries.

2.3 Complexity-theoretic Preliminaries

2.3.1 Algorithms in General

First of all, we need to be more general and use the notion of an algorithm not only in the role of an adversary but also in the role of constructions. In the case of a construction, it describes what to do in order to get from the given input to the desired output. While it is more convenient to think of a computer running the program in case of an adversary, this intuition works poorly for the construction sense. The construction is better thought of as a set of instructions by which the computer (be it a machine or human) could work. However, everything said in the first chapter about the notion still applies – it does not matter what sort of a formalization or an implementation we consider. All that is important is that the instructions could be followed in a bounded time (be it bounded in the number of steps, in actual computational time or what not), can be completed without any creativity but with an access to a random source (a.k.a. given a certain input and a certain sequence of random coins, the instructions always lead to the same result with the same amount of time or steps). All these assumptions (including the fact that we consider a random source to be available at all times which means that the algorithms are randomized) are implicit in the following chapter.

In the first chapter we used the notion of t-time algorithm. In reality, it is more common to speak of polynomial-time algorithms, which means that the number of steps t allowed is bounded by a polynomial of the length of the input or by some other specific parameter (for instance, if the input is a function, such as for the collision-resistance adversary, we often consider the input length of the hash function to be the parameter instead of the length of the description). The polynomial can be of arbitrarily large degree and can have arbitrarily large constants, and thus such an adversary may not be computable in practice. What we are more interested in, however, is the fact that all the functions that can be calculated fast for inputs of reasonable size do work in polynomial time so if we can rule out any polynomial time adversary, we can be rather sure no efficient adversary could exist at all for any larger inputs.

Formally, we say $f(x) = O(g(x))$ (f is *bounded* by g) if there exist c and k_0 such that $\forall x > k_0 \colon f(x) < cg(x)$ and we say that $f(n)$ is polynomial in n if $f(n) = O(n^c)$ for some c. For further convenience we also define $f(x) = \omega(g(x))$ to hold when $\lim_{x\to\infty} \frac{g(x)}{f(x)} = 0$. We define an adversary to be in polynomial time relative to n if t is bounded by a polynomial of n. We also sometimes say *efficient* instead of polynomial-time.

2.3.2 Oracle Machines

We also need the notion of an oracle machine. For our purposes, we say that an algorithm A is an *oracle machine* with an oracle $\mathcal{O}$ (denoted $A^{\mathcal{O}}$) if $\mathcal{O}$ calculates a function $f\colon \{0,1\}^* \to \{0,1\}^*$ and $A^{\mathcal{O}}$ can make calls to that function that are then calculated for him in a fixed number of steps by the oracle, regardless of how much time the computation would actually take. Since we work in a polynomial security model, we can directly adopt the complexity-theoretic model of an oracle working in one timestep. The notion of an oracle machine formalizes the notion of an algorithm with black-box access to a certain function. The oracle can be thought of as a module, providing certain type of functionality. For instance, the construction for h given in Theorem 2.1.1 used f as an oracle. We also note that the oracle function may even be hard or even impossible to compute. The functionality of calculating it is given to the algorithm from the outside and it does not concern the algorithm how the value for the oracle function is found. The name "oracle" even suggests it for an oracle is a person or a prophetic agency considered to be a source of superhuman knowledge in the non-technical speech. Where the oracle gets his or her knowledge is beyond the realm of reason but is usually not questioned.

We also describe what it means for an adversary to break a certain security property (such as collision-resistance). Without going too much into the mathematical formalism, we say that a *primitive* $\mathcal{P}$ is a security condition along with all the possible functions for which the security constraint has a meaning but may or may not hold. Collision-resistant hash function is a good example of a primitive – the security constraint is that of collision resistance and the family of functions is that of all the possible hash functions as it makes sense to talk about collision resistance for all of them. We denote the set of the functions associated with the primitive $\mathcal{P}$ as $\mathcal{F}_{\mathcal{P}}$ and call its elements the *implementations* of $\mathcal{P}$. We say that the adversary A *$\mathcal{P}$-breaks* the implementation $f \in \mathcal{F}_{\mathcal{P}}$ or a family of implementations $\mathcal{F} \subset \mathcal{F}_{\mathcal{P}}$ if its success probability for breaking the security property of $\mathcal{P}$ for them is greater than $n^{-\omega(1)}$ (where again n is the parameter that is usually taken to be the input length of either the adversary or the primitive being broken). The reader interested in a more formal approach is encouraged to read [12].

With this mathematical machinery in place, we can define the reductions.

2.4 Possible Formalizations of Reductions

Reingold, Trevisan and Vadhan consider seven different types of reductions in [12], starting with the most restrictive "fully black-box" to the least restrictive "free" reduction. We explore only those relevant to further discussion.

2.4.1 Fully Black-box Reductions

Simply put, the notion of a fully black-box reduction captures the simplest form of cryptographic reduction where the new primitive is constructed from the old by using the original primitive as a black box. In this case, the adversary for the original primitive is also constructed in a black-box fashion from the original primitive and the adversary for the new construction. Since we formalize the notion of a construction with a black-box access to something as an oracle machine, this gives us the following definition:

Definition 2.4.1. We say that there exists a *fully black-box* reduction from a primitive $\mathcal{P}$ to primitive $\mathcal{Q}$ if there exist polynomial-time oracle machines G and S such that

Correctness For every implementation $f \in \mathcal{F}_{\mathcal{Q}}$ we have that $G^f \in \mathcal{F}_{\mathcal{P}}$.

Security For every implementation $f \in \mathcal{F}_{\mathcal{Q}}$ and every adversary A, if A $\mathcal{P}$-breaks G^f then $S^{A,f}$ $\mathcal{Q}$-breaks f.

In the definition, G is the construction of a new primitive and S is the construction of the adversary. G is allowed to use the original implementation f and S is allowed to use the new adversary and the implementation f. We require both G and S to work in polynomial time for the constructions to be efficient. That requirement is logical because we want the reductions to be usable in the real world and because most primitives can be broken given infinite time, regardless of any oracles.

The Merkle-Damgård construction given in the beginning of this chapter is a good example of a fully black-box construction. In there, $\mathcal{Q}$ is collision-resistant hash functions of type $f\colon \{0,1\}^{n+m} \to \{0,1\}^m$, $\mathcal{P}$ is collision-resistant hash functions of type $h\colon \{0,1\}^* \to \{0,1\}^m$, G is the construction for h and S is the construction for the adversary. It might be helpful to skim through the proof of Theorem 2.1.1 again to better understand how and why a reduction of this type actually works.

Since this construction is the easiest, most reductions done in cryptography are of this type. It is however quite limiting because we require the adversary to be constructed explicitly given the new adversary and f.

2.4.2 The two Semi Black-box Reductions

The semi black-box reduction is a lot less limiting because it no longer requires an explicitly constructed adversary for the original primitive based on the adversary for the new one. All it needs is that if an adversary exists for the new scheme then one must also exist for the original. This is formalized in the following way:

Definition 2.4.2. We say that there exists a *semi black-box* reduction from a primitive $\mathcal{P}$ to primitive $\mathcal{Q}$ if there exists a polynomial-time oracle machine G such that

Correctness For every implementation $f \in \mathcal{F}_{\mathcal{Q}}$ we have that $G^f \in \mathcal{F}_{\mathcal{P}}$.

Security For every polynomial-time oracle machine A_1 there exists a polynomial time oracle machine A_2 such that for every implementation f, if A_1 $\mathcal{P}$-breaks G^f then A_2 $\mathcal{Q}$-breaks f.

We note that every black-box reduction is also a semi-black box-reduction: if there exists a polynomial time adversary A, we can take $A_1 = A$ and $A_2 = S^A$. Since both S and A are polynomial-time, it follows that so is A_2 and the implication is proved. Semi black-box constructions are however clearly more general because all we have to prove in this case is that an adversary to the original primitive $\mathcal{Q}$ exists, which may often be done without explicitly constructing it. This definition roughly corresponds to a non-constructive security proof where f is still black-box.

We also introduce a similar but even more general notion of $\forall\exists$-semi black-box reduction:

Definition 2.4.3. We say that there exists a $\forall\exists$-*semi black-box* reduction from a primitive $\mathcal{P}$ to primitive $\mathcal{Q}$ if for every $f \in \mathcal{F}_{\mathcal{Q}}$ there exists a polynomial-time oracle machine G such that

Correctness $G^f \in \mathcal{F}_{\mathcal{P}}$.

Security For every polynomial-time oracle machine A_1 there exists a polynomial-time oracle machine A_2 such that if A_1^f $\mathcal{P}$-breaks G^f then A_2 $\mathcal{Q}$-breaks f.

This allows us to supply a different construction for every possible f instead of constraining us to a "one size fits all" variant that has to work for every single implementation. It should again be clear that all semi black-box constructions are also ∀∃-semi black-box constructions. We also note that to date, reductions that don't fit the last pattern are nearly unheared of. Therefore, if one could prove that no ∀∃-semi black-box reduction can exist between two primitives then that essentially means that tools used in modern cryptology are unable to give such a construction.

We also note that the two preceeding definitions differ a little from those given by Reingold et al. The difference is in the fact that we can construct A_2 for every A_1 regardless of which f it breaks. The definitions given here better reflect common cryptographic practice and the intuition of actual cryptographers.

2.4.3 Relativizing Reductions

As noted before, the idea of proving the nonexistence of reductions of certain type arose from complexity theory, where the central question for 35 years has been "P = NP?". In 1975 Baker, Gill and Solovay used a clever trick in [1] to show that all the possible reduction types used in complexity theory during that time could not show that equality. The method they used became known as oracle separation and the reductions it prevented were named relativizing reductions.

The same idea was carried over from complexity theory into cryptography by Impagliazzo and Rudich in [8]. While the previous definitions of reductions formalized our intuitive ideas about what a reduction could possibly look like, this reduction type is more of an ingenious mathematical tool.
However, we first need to introduce the notion of a primitive existing relative to some oracle.

Definition 2.4.4. We say that a primitive $\mathcal{P}$ exists relative to an oracle Π if there exist polynomial-time oracle machines that implement $\mathcal{P}$ when given access to Π and that at least one of them is secure even when an adversary has access to Π.

This essentially formalizes the case where we just add a new base operation to the computational model we are using – besides being able to do all the normal operations in one step, we allow both the adversaries and the function constructions to make oracle calls to Π. As usual, we say that a primitive exists if we can implement it. This definition allows us to define the notion of a relativizing reduction which essentially means a reduction that holds in the presence of all possible oracles.

Definition 2.4.5. There exists a *relativizing reduction* from a primitive $\mathcal{P}$ to a primitive $\mathcal{Q}$ if for every oracle Π, if $\mathcal{Q}$ exists relative to Π then so does $\mathcal{P}$.

This definition has one clear advantage over the previous ones. Namely, it is rather easy to prove that no relativizing reductions exist between two primitives – all we need for that is to show that there is an oracle $\mathcal{O}$ such that the primitive $\mathcal{Q}$ exists relative to it but that no polynomial time implementation of $\mathcal{P}$ is secure against adversaries with an access to that oracle.

However, what makes this definition useful is the fact that it fits into the previous hierarchy. Firstly, all fully black-box constructions are relativizing. To see that, assume that there exists a black box reduction from $\mathcal{P}$ to $\mathcal{Q}$ that is not relativizing. Then there exists such an oracle $\mathcal{O}$ that $\mathcal{Q}$ exists relative to it but $\mathcal{P}$ does not. Let $f \in \mathcal{F}_{\mathcal{Q}}$ be an efficient and secure implementation of $\mathcal{Q}$ relative to $\mathcal{O}$. It then follows from the black-box reduction that there exists $G^f \in \mathcal{F}_{\mathcal{P}}$ for which there is an adversary $A^{\mathcal{O},f}$ that breaks it. Then $S^{A,f,\mathcal{O}}$ is an adversary for f that breaks it, which is a contradiction since we assumed f to be secure relative to $\mathcal{O}$. Since all fully black box reductions are relativizing, we can use the oracle separation technique to rule out the possibility of fully black box reductions. This approach has proven quite fruitful for many different important primitives. However, Those types of results can usually be extended. For that, we need to look a little downward in the hierarchy.

We note that for all relativizing reductions we can construct an equivalent $\forall\exists$-semi black-box reduction. To see that, assume we have a relativizing reduction from $\mathcal{P}$ to $\mathcal{Q}$ and consider an implementation $f \in \mathcal{F}_{\mathcal{Q}}$. If f is secure against any polynomial time adversary A_1^f, then $\mathcal{Q}$ exists relative to oracle f which also implies that $\mathcal{P}$ exists relative to f or that there exists a polynomial time implementation

$G = G^f$ (we just dont use the oracle f) of $\mathcal{P}$ for which no polynomial-time adversary A_2^f could break it.

What turns out to be more interesting, however, is the fact that in almost all cases, the reverse implication also holds – namely, that for most $\forall\exists$-semi black-box reductions there also exists a relativizing one. All that is required for that is to be able to embed any oracle into any implementation of the primitive $\mathcal{Q}$ so that the implementation would still be secure and well-formed. The proof is described in [12] and they also give the formal requirements made to the embedding. The only problem with using their approach is that they consider only binary oracles that have exactly two possible output values. Their approach can, however, be extended to cover arbitrary oracles. The problem is that we cannot hope to be able to embed any possible oracle into some primitives because their output length is usually limited. However, since our main interest is in trying to prove the non-existence of a reduction, we can get by with just embedding the separation oracle. This leads to the following theorem adopted from [4] but considered as folklore there:

Theorem 2.4.1. *Assume there is an oracle $\mathcal{O}$ and that there is a polynomial-time implementation $f \in \mathcal{F}_\mathcal{Q}$ secure relative to an oracle $\mathcal{O}$ but no polynomial time implementation $g \in \mathcal{F}_\mathcal{P}$ is secure relative to $\mathcal{O}$. Suppose further that $\mathcal{O} = \pi^f$ for a polynomial time algorithm π. Then there exist no $\forall\exists$-semi black-box reductions from $\mathcal{P}$ to $\mathcal{Q}$.*

Proof. Let f be the secure and efficient implementation of $\mathcal{Q}$ relative to $\mathcal{O}$. Assume that there exists a $\forall\exists$-semi black-box reduction and let $g = G^f \in \mathcal{F}_\mathcal{P}$ be the end result of it for f. Since, by the premises, no implementation of $\mathcal{P}$ is secure relative to $\mathcal{O}$, there exists an adversary $A_1^\mathcal{O}$ that $\mathcal{P}$-breaks g. Since $\mathcal{O} = \pi^f$ ($\mathcal{O}$ can be computed using f in polynomial time), we can convert $A_1^\mathcal{O}$ into $A_1'^f$ that does exactly the same things but uses π^f instead of $\mathcal{O}$. Since we have a $\forall\exists$-semi black-box construction, it follows that there also exists an adversary A_2^f and we can convert it back to $A_2'^\mathcal{O}$ by noting that f has to be computable in polynomial time when given access to $\mathcal{O}$. This however means that $A_2'^{\mathcal{O}}$ $\mathcal{Q}$-breaks f, which is a contradiction since we assumed that f was secure relative to $\mathcal{O}$. □

We see that this theorem is quite general. All that it requires is that we are able to embed the separating oracle into an instance of the original primitive that is also secure with respect to that oracle. This is usually rather easy to do, since we can set aside a negligible fraction of the inputs to be used for oracle calls and use the rest as we would normally.

2.5 A Proof Technique With two Oracles

Hsiao and Reyzin give an alternative method of proving there are no fully black-box reductions in [7] using two oracles instead of one. We give their result along with a sketch of a proof:

Theorem 2.5.1. *Let A and f be two oracles such that*

(a) There is a polynomial-time oracle machine T^f that implements $\mathcal{Q}$.

(b) For all polynomial-time oracle machines P, if P^f implements $\mathcal{P}$ then there is a polynomial-time oracle machine $D^{A,f}$ that breaks P^f.

(c) There is no polynomial-time oracle machine S such that $S^{A,f}$ breaks T^f.

Then there exist no fully black-box reductions from $\mathcal{P}$ to $\mathcal{Q}$.

We note that f can be thought of as advice for implementing $\mathcal{Q}$ well and A can be thought of as the adversary part of the oracle that can be used to break $\mathcal{P}$ but is useless against that one good implementation of $\mathcal{Q}$.

Proof. This theorem is essentially a corollary of the original oracle separation through relativization. Combine A and f into a single oracle (A, f). Then $\mathcal{Q}$ clearly exists relative to it since T^f implements $\mathcal{Q}$ because of (a) and no polynomial-time adversary breaks it with that oracle due to (c). The property (b) ensures that no implementation of $\mathcal{P}$ is secure and thus the result follows. □

The importance of this theorem is more in the idea here – we are allowed to give a good advice oracle that helps us with constructing a secure instance of $\mathcal{Q}$ along with the oracle meant for breaking $\mathcal{P}$. This advice oracle then ideally allows us to prove unconditional irreducibility theorems for even the primitives we don't normally know how to implement well. Other than that, the theorem offers relatively little new.

2.6 The Approach of this Thesis and Related Work

As we have shown thus far, all that would be needed to rule out fully black-box constructions is an oracle that could be used to break chain-resistance while leaving at least one family of functions collision-resistant. Buldas and Saarepera [4] demonstrated that no collision-resistant hash function could be proved to be chain-resistant by fully or semi black-box reductions. The oracles they used are of little help to us, however, since they rely on the same function being broken for both chain and collision resistance. The article still suggests that oracle separation could nonetheless be used to rule out the construction of a chain resistant function from a collision resistant one. We explore that possibility by trying to construct an oracle based on a suitable hash tree. How it is done exactly is covered in the next chapter.

We also note some other related work. Jürgenson and Buldas show in [2] that black-box constructions cannot give a collision-resistant function based on chain-resistant functions. Their result is essentially the other direction of what we are trying to prove. For practical considerations, Buldas and Laur prove in [3] that the hash functions that are chain-resistant need not even be one-way.

3. The Hash Tree Oracle and a Pair-checking Adversary

3.1 The Construction Idea for the Separation Oracle

As mentioned at the end of the previous chapter, our approach is to try to construct a separation oracle that could rule out constructions of chain-resistant functions from collision resistant ones. For that we need an oracle that breaks all implementations of chain-resistance while leaving at least one family of hash functions secure in the collision-resistance sense. The first thing we note is that chain-resistance is quite a complex property. It is hard to think of any useful information that the oracle could give that would help break it other than the actual root value and certificates. There is one natural candidate for an oracle that can do that – namely one that constructs a tree from a large amount of inputs and then returns the root value and certificates based on that tree.

We now consider the oracle as constructing a tree from all the possible inputs $x \in \{0,1\}^n$ that could be sent to be timestamped. We can formalize the oracle $\mathcal{O} = (\mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3)$ such that $\mathcal{O}_1(H)$ returns the root value of that tree constructed for the hash function H and $\mathcal{O}_2(x, H)$ gives a certificate (n, z) for x by taking the path from that tree starting from the input x. It is clear that such an oracle will break every hash function in the chain-resistance sense with probability 1. Inspired by the theorem with two oracles in the previous chapter, we also add a third part

$\mathcal{O}_3$ to the oracle which would implement a truly randomly chosen function from some well-chosen hash function family. All we would need in this case is to show that the hash function supplied by $\mathcal{O}_3$ is hard to break. Due to it being computed by the oracle the hash function that we deem unbreakable does not have to be polynomial-time. It also means we get control over information flow in the oracle, which is to say, the oracle knows how much info about the function leaks or would leak to the adversary when it constructs a certain type of tree since it can observe what oracle calls would be made in its computation.

We note that this representation for the oracle is clearly not optimal in the sense of the lenghts of its inputs and outputs. We could modify them so that the outputs they currently give could be computed in polynomial time from the ones they would give then. For instance, $\mathcal{O}_2$ can give out only the corresponding z_1 and n_1 of the certificate, as the next entry can then be constructed by computing $H(x, z_1)$ or $H(z_1, x)$ (depending on n_1) and we can call the oracle again and again until we end up in the root. There are a few other tricks that could be used. However, these shorter representations would only be useful if we considered embedding the oracle into a function, which we would need for ruling out semi black-box constructions. This thesis is mainly concerned with fully black-box constructions and as such leaves the possibility of an embedding as a future problem to be solved once this oracle is indeed shown to give the desired separation.

There are, however, severe limitations on using this type of oracle. We now try to describe what they seem to be and how they could possibly be avoided.

3.2 How not to Construct a Separation Oracle

The approach suggested in the preceeding section initially sounds very promising. However, Buldas managed to prove (as a yet unpublished result) that if the oracle indeed gives out the root of a full tree (a tree with all the possible inputs x) from

$\mathcal{O}_1$, such an oracle can always be exploited to find a collision. We state the result as a theorem:

Theorem 3.2.1. *Let $h\colon \{0,1\}^n \to \{0,1\}^m$ be a hash function. Then there exists a polynomial time oracle machine construction for $H\colon \{0,1\}^{4n} \to \{0,1\}^{2n}$ such that if a full tree is built from H^h, its root value $r = (r_1, r_2)$, $r_1, r_2 \in \{0,1\}^n$ will be a collision for h.*

Proof. We show that $H^h(x_1, x_2, y_1, y_2)$ $(x_1, x_2, y_2, y_2 \in \{0,1\}^n)$ can be constructed as follows:

- Check if x_1 and x_2 form a collision for h. If they do, return (x_1, x_2).
- Check if y_1 and y_2 form a collision for h. If they do, return (y_1, y_2).
- If neither condition was satisfied, return 0^{2n}.

It is clear that if a collision is ever presented to H^h as either left or right input, it will also return a collision as output – it can be the same collision but it may also be the collision passed as the other input. Since we are presented with a full tree, every possible $2n$-bit string is given as input somewhere. Since there is at least one collision, a string encoding that must also be given as input from somewhere and it follows that a collision will also nessecarily be returned as the root value because there is a path of H^h leading from the collision down into the root. $\square$

This theorem means that an oracle that constructs the full tree can always be exploited. We note that we do not need to break the chain-resistance property with probability one. This means that we are allowed to fail to produce a certificate for some inputs x for $\mathcal{O}_2$ as long as we can produce a certificate with a non-negligible probability. This means that the tree does not need to be full, but it still needs to be quite large – it needs to use at least a polynomial fraction $\frac{1}{p(n)}$ of all possible inputs.

3.3 Extending the Construction

The previous construction leaves some room for generalization. Suppose we check more than one pair per each input, but still pass on our findings so if a collision is ever found, it will propagate to the root. This leaves us with a question of what inputs to check.

After some consideration we can formulate our question in terms of graph theory. Suppose we are trying to construct $H\colon \{0,1\}^{2k} \to \{0,1\}^k$ that will find collisions for $h\colon \{0,1\}^n \to \{0,1\}^m$. Since we are trying to mimic the previous construction, assume that for each $x \in \{0,1\}^k$ the function has a set of pairs in h it will check for and if it finds a collision it will pass that same value x down. We can then construct a bipartite graph. Let P_n be the set of unordered pairs from $\{0,1\}^n$ (so $|P_n| = 2^{n-1}(2^n - 1)$) and let $G = (\{0,1\}^k, P_n, E)$ be the bipartite graph such that $(x,(y_1,y_2)) \in E$ iff H checks the pair (y_1,y_2) when given x as input. What we would like is for every subset of $\{0,1\}^k$ that has at least $\frac{2^k}{p(n)}$ elements to have nearly all the elements of $\{0,1\}^{2n}$ as its neighbours (so at least one collision would be among the neighbours). That would mean that no matter what inputs are given to that tree, assuming there are at least $\frac{2^k}{p(n)}$ of them, a collision will always be found. There is one additional constraint – namely, we can check only a polynomially bounded number of pairs, as H is expected to work in polynomial time. It turns out that just such types of graphs have been considered before in other applications.

3.4 Disperser Graphs

Definition 3.4.1. We call a bipartite graph $D = (V_1, V_2, E)$ a (K,ϵ)-*disperser* if the neighbour set $N(U)$ of every $U \subset V_1$ with cardinality K has at least $(1-\epsilon)|V_2|$ elements in it.

In our case, we are looking for a $K = \frac{2^k}{p(n)}$ disperser graph with as small ϵ as we can possibly have. It turns out that there are well-known lower bounds on all the parameters of disperser graphs.

Dispersers are mainly used as theoretical tools for randomized complexity classes or for extracting randomness from a weak source. They are generally considered alongside extractors, which serve a similar but stronger role. Both types of graphs are often used in complexity theory and cryptography and there are numerous good surveys about their properties, consntructions and bounds (see Shaltiel [13] for one).

Let D be the average degree of a vertex in V_1. One thing we have to note straight away is that for us to have any hope of always covering enough of V_2, we need $KD \geq (1-\epsilon)|V_2|$ since otherwise there simply are not enough neighbours. Since most of the time some neighbours overlap, this would be an idealistic scenario. As it turns out, there are much more strict bounds for the parameters. In fact, Radhakrishnan and Ta-Shma give the following theoretical bounds in [11]:

Theorem 3.4.1. *Suppose that $G = (V_1, V_2, E)$ is a (K, ϵ)-disperser with $N = |V_1|$ and $M = |V_2|$. Let D be the average degree of a vertex in V_1.*

(a) Assume that $K < N$ and $D < \frac{(1-\epsilon)M}{2}$ (so G is not trivial). if $\frac{1}{M}\epsilon < \frac{1}{2}$ then $D \geq \frac{1}{\epsilon} \log \frac{N}{K-1}$.

(b) Assume that $K \leq \frac{N}{2}$ and $D \leq \frac{M}{4}$. Then $\frac{DK}{M} \geq c \log \frac{1}{\epsilon}$ for some c.

We try to apply these bounds to see how far the previously presented adversary idea could take us. Let $q(n)$ be the polynomial by which D is bounded. From (a) we then find that $q(n) \geq c\frac{1}{\epsilon}\log(n)$ for some constant c and thus $\epsilon \geq \frac{c\log(n)}{q(n)} \geq \frac{1}{q'(n)}$ for some polynomial $q'(n)$. This means that we can guarantee that all but $\frac{1}{q'(n)}$ of all the possible pairs are covered. We now turn to study the structure of the hash functions and their collisions to show how well this approach could work.

3.5 The Structure of Hash Function Collisions

3.5.1 Collision Graphs

It should be clear that we can view collisions as edges in a graph with the vertex set being the domain of the hash function. We formalize that with the following definition:

Definition 3.5.1. Let $h\colon \{0,1\}^n \to \{0,1\}^m$ be a hash function. We call the graph $G_h = (\{0,1\}^n, E)$ a *collision graph* for h if $(x_1, x_2) \in E$ precisely when x_1 and x_2 form a collision.

We now formally try to describe the structure of collision graphs with simple combinatorial arguments.

Lemma 3.5.1. *Let G_h be a collision graph for $h\colon \{0,1\}^n \to \{0,1\}^m$. Then for every $x \in h(\{0,1\}^n)$, the vertices in $h^{-1}(x)$ form a clique in G_h.*

Proof. Choose $x_1, x_2 \in h^{-1}(x)$, $x_1 \neq x_2$. Then $h(x_1) = x = h(x_2)$ and so they form a collision. □

We note that a single vertex without any neighbours also constitutes a clique.

The following lemma completely describes the structure of collision graphs.

Lemma 3.5.2. *The collision graph G_h for $h\colon \{0,1\}^n \to \{0,1\}^m$ is the union of at most 2^m vertex-disjoint cliques. Also, for any graph G of 2^n vertices and the previous property we can construct a h' such that G is the collision graph of h'*

Proof. There are at most 2^m elements in $h(\{0,1\}^n)$. The original of each of them forms a clique and no vertex can belong to two different originals so the cliques have to be vertex-disjoint. The second part is obvious, as we can index the cliques with values from $\{0,1\}^m$ and then construct the h' so that $h'(x)$ would return the index of the clique the vertex x is in. □

3.5.2 Edges to Check in the Best Case

We are interested in the question of how many edges we have to check before we can be sure there is at least one collision among them.

Lemma 3.5.3. *Let $h\colon \{0,1\}^n \to \{0,1\}^m$ be a hash function. Then there is at least one $x_0 \in \{0,1\}^m$ such that $|h^{-1}(x_0)| \geq 2^{n-m}$.*

Proof. Assume that the statement does not hold. Then for all $x \in \{0,1\}^m$ we have that $|h^{-1}(x)| < 2^{n-m}$. But then $|h^{-1}(\{0,1\}^m)| < 2^{n-m} \times 2^m = 2^n$ $\square$

Corollary 3.5.4. *The collision graph G_h of $h\colon \{0,1\}^n \to \{0,1\}^m$ has at least one clique of size 2^{n-m}.*

We now state a classical theorem of graph theory proved by Turan in 1941

Theorem 3.5.5. *(Turan) Let $G = (V, E)$ where $|V| = n$ and G contains no clique of size $r + 1$ or larger. Then*

$$|E| \leq \left\lfloor \left(1 - \frac{1}{r}\right) \frac{n^2}{2} \right\rfloor$$

and equality can be achieved.

This allows us to prove the following:

Theorem 3.5.6. *Let $h\colon \{0,1\}^n \to \{0,1\}^m$ be a hash function. Then we can guarantee that we find a collision by checking for 2^{n+m} cleverly chosen pairs.*

Proof. By Turan theorem there exists a graph with the vertex set $\{0,1\}^n$ such that it has no cliques of size 2^m but has $|E| = \left\lfloor \left(1 - \frac{1}{2^{n-m}-1}\right) \frac{2^{2n}}{2} \right\rfloor$ edges. If we check all the pairs corresponding to edges in the complement of that graph, the collision graph can not be formed from the remaining edges because by the previous lemma, it has to have a clique of 2^{n-m} vertices but no such clique can exist due to the original graph chosen being free of them. The complement of that graph has $\frac{1}{2}2^n(2^n - 1) - |E| < 2^{n+m}$ edges so we can do it by checking 2^{n+m} pairs. $\square$

We should note that while in general, any discrete function with domain smaller than the range is considered a hash function, it is generally assumed that the difference $n - m = \omega(\log(n))$ because if it is smaller, then the "secure" constructions are usually trivial but also hard to use in practice. We also make the same assumption in our work and henceforth consider the difference $n-m$ to be bounded from below as mentioned.

Considering that using the disperser approach, we can cover all but a polynomial fraction of the pairs, there is always a possibility of the set induced by the disperser covering those 2^{n+m} cleverly chosen pairs and thus finding a collision. This means that the disperser construction cannot be ruled out by simple combinatorial claims, since there is a theoretical possibility of a K element subset always covering those vertices that guarantee us a collision.

However, constructing a disperser that does that is a different story entirely. It is known that there are dispersers with $D = \Theta(\frac{n}{\epsilon})$ and with $\log \frac{KD}{M} = \log\log \frac{1}{\epsilon} + c$ and there are explicit constructions that come pretty close to these bounds (for instance the one given in [15]), covering a certain set (or actually one of a a family of sets as we can permute the vertices in the Turan construction) is not guaranteed by the usual constructions.

3.5.3 Edges to Check in the Worst Case

We however note that the Turan construction is a good case – if we can choose the pairs ourselves, we can get by with asking for just 2^{n+m} of them. We still have not answered the question of how many pairs we would need to check in the worst case. The following theorem and its corollary addresses that question.

Lemma 3.5.7. *Let* $h\colon \{0,1\}^n \to \{0,1\}^m$ *be a hash function. Then there are at least* $2^{n-1}(2^m-1)$ *collisions.*

Proof. We know the collision graph is composed of vertex-disjoint cliques and that a clique of size n has $\frac{1}{2}n(n-1)$ edges. Let $n_i = |h^{-1}(i)|$ where $i \in \{0,1\}^m$ so $\sum_{i\in\{0,1\}^m} n_i = 2^n$. Then there are $\sum_{i\in\{0,1\}^m} \frac{1}{2}n_i(n_i-1) = \frac{1}{2}\left(\sum_{i\in\{0,1\}^m} n_i^2 - 2^n\right)$ collisions and the formula is minimized when n_i are chosen as equal as possible. If h maps exactly 2^{n-m} elements to each $x \in \{0,1\}^m$ then there are exactly

$2^{n-m}\left(\frac{1}{2}2^m(2^m-1)\right) = 2^{n-1}(2^m-1)$ collisions. □

Corollary 3.5.8. *Let* $h\colon \{0,1\}^n \to \{0,1\}^m$ *be a hash function. Then there can be at most* $2^{n-1}(2^n - 2^m)$ *pairs that do not form a collision.*

We would therefore need to cover all but roughly $2^{-(n-m)}$ fraction of the inputs. Since $n-m = \omega(\log n)$, this means a $\frac{1}{n^{\omega(1)}})$ fraction and thus the disperser approach does not get us far enough, because we leave a polynomial fraction uncovered and this is less than what is needed.

3.6 Infeasibility of the Disperser Approach

We now take a step back and try to understand why extending the construction fails. We note that our approach is essentially the following – check as many pairs as you can in the tree and hope you catch one. If you do, just pass it down to the root. Since the tree is of exponential size, the hope of finding one is quite large, so it is a reasonable trick to try. So why doesn't it work? Maybe the approach of just checking pairs is too simplistic. We note that for these types of adversaries we can replace the oracle for $h = \mathcal{O}_3$ given to A with a "collision oracle" $c_h\colon \{0,1\}^{2n} \to \{0,1\}$ such that $c_h(x,y) = 1$ precisely when $h(x) = h(y)$. We now explore the boundaries of adversaries with just that oracle.

In this case, the problem for constructing the oracle parts $\mathcal{O}_1$ and $\mathcal{O}_2$ becomes simple. All we have to ensure is that we form the tree in such a way that we can always answer 0 to the oracle queries presented to us within it. If we can ensure that we answer few enough queries and we answer all of them negatively, then it follows rather directly that collision-resistance of $h = \mathcal{O}_3$ is preserved. It turns out that if we only consider the collision oracle, this problem is (nearly) equivalent to trying to find a good enough disperser.

3.6.1 Adjusted Problem Statement

We begin with a definition:

Definition 3.6.1. We say that $H^c \colon \{0,1\}^{2k} \to \{0,1\}^k$ is a *Hash adversary* for a hash function family χ of $h \colon \{0,1\}^n \to \{0,1\}^m$ if it is computed by a deterministic oracle machine that makes at most a polynomial number $p_H(k)$ of calls to the collision oracle c_h during any call $H(x)$ where $x \in \{0,1\}^{2k}$.

We stress that we consider deterministic machines for hash adversaries. This is only natural because we expect H to compute a function.

We now prove a somewhat suprising result showing that our problem can be reduced to a much simpler one.

Theorem 3.6.1. *Let $\mathcal{F}$ be a family of hash functions $h \colon \{0,1\}^n \to \{0,1\}^m$. Assume that for any Hash adversary H and for all but an ϵ fraction of $h \in \mathcal{F}$ there exists a subset K of cardinality $\delta 2^{2k}$ such that for all $x \in K$ no call to c_h made by $H(x)$ covers an actual collision for h. Then we can construct a tree with at least $\delta 2^{2k}$ inputs for any hash adversary H' such that no collisions are checked during its execution. The reverse also holds.*

Proof. Let $H \colon \{0,1\}^{2k} \to \{0,1\}^k$ be any hash adversary. We construct a full Merkle tree of H^{c_0} such that every possible input $x \in \{0,1\}^{2k}$ is presented to it exactly once. We use the dummy oracle c_0 that always returns 0. Based on that tree, we define $H'(x)$ to do all the calculations and calls to the oracle c done by H^{c_0} on the path from $H(x)$ down to the root. Since H makes at most a polynomial number of calls to the oracle and the path is of length k, H' is also a hash adversary. We can therefore extract a $\delta 2^{2k}$ cardinality subset K for which H' will not ask for any actual collision of h from c (for all but an ϵ fraction of $h \in \mathcal{F}$). We construct the hash tree for H as the union of all the paths in the Merkle tree that begin from vertices in K. Since no collisions are found on these paths due to the construction of the set K, these paths are the same as those in the original Merkle tree constructed with the dummy oracle c_0. This means that their union indeed forms a proper tree and since no path finds a collision, the tree constructed as their union also fails to find one.

The reverse is actually trivial: Let $H\colon \{0,1\}^{2k} \to \{0,1\}^k$ be a Hash adversary and assume that for some $h\colon \{0,1\}^n \to \{0,1\}^m$ we can find a tree with at least $\delta 2^{2k}$ different inputs and during which no collisions are found. If we just take the set of inputs given to it, we have the required subset of cardinality K for H. □

This theorem essentially says that instead of trying to construct a tree, we should concentrate all our efforts to just finding the subset of input pairs with the required size and property, because once we can do that for any hash adversary, we can also construct the tree. This removes one dimension of complexity from our problem and allows us to look at it in terms of graph problems yet again. The proof was carried out for full inputs of length $2k$ but if we can construct a tree from a polynomial fraction of them, it is bound to contain a polynomial fraction of different single inputs of length k as well.

3.6.2 Infeasibility Results

We know from before that the disperser graphs fail to give us such a construction (at least with simplistic attempts). It turns out that a probabilistic argument shows that we cannot do much better with other means either:

Theorem 3.6.2. *Assume that $H\colon \{0,1\}^{2k} \to \{0,1\}^k$ is a hash adversary with an oracle c_h for h where $h\colon \{0,1\}^n \to \{0,1\}^m$ is chosen uniformly from a family of hash functions $\mathcal{F}$. We also assume that for every pair $(x,y) \in \{0,1\}^{2n}$, $x \neq y$ the probability that $c(x,y) = 1$ is $2^{-\omega(\log(n))}$. Then the probability of not being able to construct a tree that doesn't show any collisions is negligible.*

Proof. Let $N(x_1,x_2)$ be the number of different inputs of H on which it checks for the collision $c(x_1,x_2)$. It is clear that $\sum_{(x,y)\in\{0,1\}^{4n}} N(x,y) \leq p(n)2^{2k}$ since there are a total of 2^{2k} different possible inputs and for each input the number of calls to c_h is bounded by a polynomial $p(n)$. We calculate the expected value of the number of inputs N_p that will lead to a collision being detected in their branch:

$$\begin{aligned} E[N_p] &\leq \sum_{h\in\mathcal{F}} \Pr[h] \sum_{(x,y)\in P_n} N(x,y)[h(x)=h(y)] = \\ &= \sum_{(x,y)\in P_n} N(x,y) \sum_{h\in\mathcal{F}} \Pr[h][h(x)=h(y)] = \\ &= \sum_{(x,y)\in P_n} N(x,y) 2^{-\omega(\log(n))} \leq p(n) 2^{2k-\omega(\log(n))} \ . \end{aligned}$$

Where $[\cdot]$ is the Iverson symbol (that is 1 when the predicate within the brackets is true and 0 otherwise) and P_n is the set of unordered pairs from $\{0,1\}^n$.

Using the Markov inequality we get that $\Pr[N_p \geq 2^{2k-1}] \leq p(n)2^{1-\omega(\log(n))}$ so a subtree with at least half of all the different inputs fed to it that fails to find a collision can be found for all but a $p(n)2^{1-\omega(\log(n))}$ fraction of h. Using the previous theorem now gives us the promised result with $\delta = \frac{1}{2}$. □

What this theorem roughly means is that given any fixed hash adversary, the oracle can avoid revealing the collisions to it directly. It does not, however, rule out that from such trees we could somehow deduce the collisions indirectly. We can extend the construction given in the previous theorem to rule out that as well:

Theorem 3.6.3. *Under the assumptions of the previous theorem, the probability of finding a collision can be made negligibly small.*

Proof. Let 2^{-d} be the collision probability in the previous proof. For each $h \in \mathcal{F}$ we randomly chose a set $\mathcal{F}_h \subset \mathcal{F}$ of size $2^{0.5d}$ and try to avoid all of their collisions instead of just those for h itself. The probability of randomly choosing a collision for one of them from the set of all the possible pairs is $2^{-d} \cdot 2^{0.5d} = 2^{-0.5d}$ and if $d = \omega(\log n)$ then so is $0.5d$. This means that the argumentation of the theorem will still go through and that we can still avoid the set of forbidden pairs for all but a superpolynomial fraction of $h \in \mathcal{F}$. This however also means that each such avoiding tree could be valid for any of the $2^{0.5d}$ possible hash functions in $\mathcal{F}$ instead of just one and considering the construction and size of that set, the average chance of guessing a collision correctly only knowing the set $\mathcal{F}_h$ that h belongs to it is still negligible. $\square$

This means that an adversary that only uses the approach of checking pairs of input values for collisions can always be fooled. As such, it rules out one rather simplistic approach for the adversary trying to exploit the oracle. We now turn to see if our approach could be extended or improved in any way.

4. Other Possible Approaches

By now we have shown that it is hard to exploit the tree oracle by just checking pairs for collisions. However, there are more complicated things that the adversary could do. We now rule out a few other seemingly promising approaches.

4.1 The Equality Oracle

We note that there is a lexicographic and a rather natural total ordering of bit-strings that corresponds to the order of natural numbers if the strings are interpreted as numbers in base 2. One of the first ideas about finding collisions that Buldas originally had (also an unpublished result) was to construct a hash adversary that could find the maximum and minimum input that produced a given output when the full hash tree was used. All that was essentially needed for that were checks in the form $h(x) = y$, $x \in \{0,1\}^n, y \in \{0,1\}^m$. If we formalize that in the form of an oracle $e_h(x, y)$, the same argument as presented for the collision oracle can be carried out nearly word to word to show that this approach has as little potential as our previous one, since as before, we can nearly always find trees that always answer 0 to the query. In this case the lack of potential does not follow as easily though. We note that it would take $2^m - 1$ queries of this type that were answered negatively to determine the value $h(x)$. Therefore there is a possibility of the adversary still finding out the value of $h(x)$ during the tree. However, the argument given for c_h in theorem 3.6.3 that looked at collisions for a large set of randomly chosen $h' \in \mathcal{F}$ instead of a single h could again be used rather effectively to rule out the equality oracle giving enough useful information.

4.2 The Greater-than Oracle

The approach of finding the minimum or maximum of some values gives us another idea of what to try to rule out. Suppose we base our construction of H on questions of the form $h(x) > h(y)$ where the order is the same natural lexicographic order as mentioned before. In that case we can also construct a binary oracle $g(x, y)$ to answer queries of this type. There is one very important difference between this oracle and the previous two considered. In the collision and equality oracles, the answers were clearly asymmetric in the amount of information they gave away. That meant that 0 was a "safe" answer and if we could avoid ever answering 1, that nearly guaranteed that the oracle could not be exploited. In this case, both answers 0 and 1 give out nearly equal information so always answering 0 is just as dangerous as always answering 1.

This means that we have to recheck all our steps for the collision oracle. We note that theorem 3.6.1 essentially relies on our ability to "fake" safe answers so we could get a static tree structure. As we remember from the theorem 3.2.1, if we honestly answer the queries in the full tree then the adversary can guarantee himself a collision in the root. We also note that the greater-than oracle can be used to emulate the collision oracle as $c(x, y) = 1$ happens exactly when $g(x, y) = 0$ and $g(y, x) = 0$. This means that to use the same technique, we have to find a way to fake the oracle answers consistently enough that a "correct" subtree where all the answers were correct could be extracted but yet falsely enough that the root would not give the adversary a collision. This is by no means a trivial task.

There is however a way to do so. We extend $h\colon \{0,1\}^n \to \{0,1\}^m$ into a larger range to get $h'\colon \{0,1\}^n \to \{0,1\}^{m+n}$ such that the most significant bits of $h'(x)$ correspond to the output of $h(x)$ but the least significant ones are chosen randomly with a constraint that h' is injective so no output corresponds to two different inputs. We then answer $g(x, y)$ queries based on h' instead of h. Clearly, we have to clean the tree of all the answers that say $g(x, y)$ when $h(x) = h(y)$ but this is essentially the same as clearing the tree of all the queries where $c(x, y)$ would be equal to 1.

However a small problem remains. Namely, the argument used to prove that the answers given by the oracle do not uniquely determine the collisions by some indirect means does not work as it did before. The original idea needs to be modified a little. Instead of choosing random hash functions into $\mathcal{F}_h$, we choose the other hash functions in such a way that consistent answers can still be given rather easily for the greater than. To be precise, we try to avoid answering queries such that $h(x)$ and $h(y)$ differ only in the last $0.5d$ bits where 2^{-d} is the collision probability. Essentially the same argumentation as before could then be used. Intuiton behind this approach is that instead of h we simply consider a hash function $h''\colon \{0,1\}^n \to \{0,1\}^{m-0.5d}$ for which we elliminate the collisions in the tree. Since we remove any oracle queries that distinguish between collisions within a class of size $2^{0.5d}$ output values, it is easy but rather technical to prove that even knowing all the information given out by the oracle the chance of finding a collision is still negligible.

4.3 Greater-than-by-value Oracle

We note that there is one more natural binary oracle we could try to fake: the one for queries of type $h(x) > y$. This oracle has one strength over the others: while it would take $2^m - 1$ queries to know the actual value of $h(x)$ using the equality oracle and even more for the others, it only takes roughly m queries to find the value using this one. This means that we could easily interchange between a binary oracle of this type and a full oracle for h since we can compute one from the other in polynomial time. Therefore, if we could learn to somehow fake this oracle in the same sense as we have faked the three previous ones, it would give us a full proof of the separation we have been seeking. It should then be no suprise that this oracle seems to be much harder to fake and we have not discovered a good means of doing so. The approach used for greater than does not work because if each H tries to query an exact value of $h(x)$, it can get it with m queries and there is no legal way to prevent that.

4.4 Using many Oracles in one Construction

We note that we can also rule out all the constructions that use a combination of the first tree oracles[*] – the last step in theorem 3.6.2 can be changed to use $\delta = \frac{4}{5}$. Then we can choose the sets separately for all three such that none of them gives enough information and then take their union which is of size at least a fraction $\delta' = \frac{1}{5}$. This can be done even after the tricks used in theorem 3.6.3 and its analogues have been taken into account. Therefore, this thesis rules out any adversary constructions that work based only on these three oracles and do not use any additional types of info. This seems to rule out most of the simple ways of exploiting the oracle, so if the tree oracle does not work, the adversary that can use it to break collision-resistance for $\mathcal{O}_3$ has to be fairly clever.

4.5 Possible other Constructions of Oracles

The work presented up to now leaves the possibility of oracle separation in a rather ambiguous state – the proposed oracle seems to be hard to exploit, but it seems to be nearly impossible to rule out an adversary construction that nonetheless does so. There is one more approach that could lead to some positive results. We note that for any possible function presented to the oracle, the probability of it giving any information about the collisions of h supplied by the oracle taken over all the possible inputs nearly always has to be negligible – if it is not, we could do without an oracle by simply choosing a random input, using the function on it and trying to deduce a collision based on that. We essentially proved this fact

[*]Of course, if we note that the collision oracle can be emulated with the greater-than oracle, we can only get by with two oracles and the argument would be somewhat simpler

for collision checking adversaries in theorem 3.6.2 and then extended it to other oracle types. This approach might be slightly more general in allowing for more complex hash adversaries but it leaves a problem of constructing a tree, since the approach we took in this thesis does not work any more or has to be heavily modified at the very least. It thus seems that the problem needs more complicated mathematical machinery, perhaps that of information theory and of Kolmogorov complexity but perhaps also of advanced complexity and combinatorial theory. However, there may be a better choice for an oracle that could lead to a easier separation. We now briefly discuss a few possible alterations to the current scheme.

We first note that the oracle has to be a well-defined function. It can, however, be highly dependent on its inputs. For instance we may vary the polynomial fraction of inputs we give out based on the working time of the function given as input. This may allow us to better limit the number of possible inputs seen. However, this will probably not help much.

There are certain inherent flaws in the oracle model we have been studying. The major one is that it can force us to give out exponential amounts of information during the whole calculation of the tree. This causes us theoretical problems because there is no way to rule out an exponential amount of it being transmitted through the root value since a polynomial output may carry information about an exponential sized set. This makes it nearly impossible to rule out many oracle calls eventually helping to find a collision. There may be ways of showing that any information about an exponential sized set that the root gives could not be reliable or that it could not be gathered at all. However, the current model seems to present no obvious ways of doing it.

The tree model is good if we want to avoid showing collisons for the function the tree is constructed from. However, we only have to avoid showing collisions for just one function – the one provided by $\mathcal{O}_3$. This means that we might want to abandon the tree approach and instead just concentrate on certificates that do not have to form a tree when viewed together. This removes one rather restricting constraint, but does not really solve any problems, since there seem to be no obvious good ways of constructing a root value without avoiding the problem of possibly exponential information.

There seems to be one way that clearly solves the problem with $\mathcal{O}_1$ giving too much info – namely, choosing it randomly. The best way to do that for a given $H\colon \{0,1\}^{2k} \to \{0,1\}^k$ is to uniformly generate a k element long hash chain string with uniformly chosen inputs its root value. It can, in fact, be shown that if the root value is chosen that way, a certificate exists for most inputs with rather high probability. Also, in this case, we clearly need not worry about too much information leaking from $\mathcal{O}_1$. However, there may be problems with unwanted information leaks from $\mathcal{O}_2$ since the path structure of the certificates may then reveal information that is hard to control. Because of that this model seems to be much harder to analyze than the one we mainly studied.

Conclusion

We tried to show that no black-box constructions could exist that would give a hash function secure for time-stamping from a hash-function that is collision-resistant. For that we studied an oracle that constructs a large hash tree, gives out its root value and then outputs certificates according to the tree structure. Since breaking the time-stamping property requires the tree to be of exponential size, it is hard to avoid giving the adversary the capacity of performing exponential amounts of work within that tree.

We manage to show that this oracle does not seem to be easy to exploit for finding collisions. It is known from before that if a full tree is constructed inside the oracle then it can be used to break collision-resistance for all hash functions. We try to extend the construction used to find collisions in that proof and conclude that the simplistic approach of just checking pairs of inputs for collisions is not enough to find a collision with just one oracle call. We then rule out a few other simple approaches. Namely we prove that only checking if the hash function gives a certain output for a given input and checking if one input gives a larger hash value than the other will also be insufficient, even when used together in the hash adversary.

We also briefly discuss ways of altering the oracle that may make the proofs easier in some respects. However, we conclude that the model currently used seems to be the easiest to study and that while the other models eliminate some theoretical problems, they give rise to other and more complicated ones.

Kollisioonivabadel räsifunktsioonidel põhinevate piiranguteta ajatempliskeemide võimalikkusest

Magistritöö (20 AP)

Margus Niitsoo

Käesolevas töös uurime piiranguteta ajatempliskeemi jaoks turvaliste räsifunktsioonide konstrueerimise võimalusi kollisioonivabadest räsifunktsioonidest. Kasutades Harberi ja Stornetta poolt loodud ajatembeldusskeemi ning Buldase ja Saarepera poolt selle jaoks konstrueeritud turvatingimust uurime nn. musta kasti konstruktsioonide võimatuse tõestuse võimalikkust. Kuna võimatuse tõestuse lihtsaim variant on oraakliga eraldus, keskendumegi just ühe selle eralduse jaoks sobivana tunduva oraakli omaduste ja võimaluste uurimisele.

Me eeldame, et oraakel konstrueerib räsipuu, väljastab puu juurväärtuse ning annab seejärel sellest puust lähtuvalt ajatemplisertifikaate. Me tõestame, et kui oraakli argumendiks olev musta kasti meetodil koostatud räsifunktsioon ainult algse räsifunktsiooni kollisioonipaare kontrollib või nn. suurem-kui predikaati kasutab, ei saa seda oraaklit kasutada kollisioonide leidmiseks . Töö tulemused annavad lootust, et nimetatud oraakel on tõepoolest eralduseks sobiv ja lubavad oletada, et sarnaste oraaklite edasine uurimine võib lõpuks probleemi lahenduseni viia.

Bibliography

[1] Baker, T. J. , Gill, J., Solovay, R. "Relativizations of the P=NP question", SIAM Journal of Computing 1975 (4), pp 431-442, 1975

[2] Buldas, A, Jürgenson, A. "Does Secure Time-Stamping Imply Collision-Free Hash Functions?", Provable Security, pp 138-150, 2007

[3] Buldas, A. , Laur, S. "Do Broken Hash Functions Affect the Security of Time-Stamping Schemes?", Applied Cryptography and Network Security, pp 50-65, 2006

[4] Buldas, A. , Saarepera, M. "On Provably Secure Time-Stamping Schemes", Advances in Cryptology - ASIACRYPT 2004, pp 500-514, 2004

[5] Damgård, I. "A Design Principle for Hash Functions", Advances in Cryptology - CRYPTO '89 Proceedings, pp. 416-427, 1989

[6] Haber, S. , Stornetta, W.-S. "Secure Names for Bit-Strings", ACM Conference on Computer and Commuinications Security, pp 28-35, 1997

[7] Hsiao, C.-Y. , Reyzin, L. "Finding Collisions on a Public Road, or Do Secure Hash Functions Need Secret Coins?", Advances in Cryptology - CRYPTO 2004, pp 92-105, 2004

[8] Impagliazzo, R., Rudich, S. "Limits on the the provable consequences of one-way permutations", Proceedings of the 21st ACM Symposium on the Theory of Computing, pp 44-61, 1989

[9] Merkle, R. C. "A Certified Digital Signature",Advances in Cryptology - CRYPTO '89 Proceedings, pp. 218-238, 1989

[10] Merkle, R. C. "Protocols for public-key cryptosystems.", Proceedings of the 1980 IEEE Symposium on Security and Privacy, pp.122-134, 1980

[11] Radhakrishnan, J. , Ta-Shma, A. "Bounds for dispersers, extractors, and depth-two superconcentrators", SIAM Journal on Discrete Mathematics, vol 13, issue 1, pp 2-24, 2000

[12] Reingold, O. , Trevisan, L. , Vadhan, S. "Notions of Reducibility between Cryptographic Primitives", Theory of Cryptography, pp 1-20, 2004

[13] Shaltiel, R. "Recent Developments in Explicit Constructions of Extractors.", Bulletin of the EATCS, vol 77, pp 67-95, 2002

[14] Stinson, D. "Cryptography : theory and practice", 3rd edition, Chapman & Hall, 593 pg, 2006

[15] Ta-Shma, A. "Almost optimal dispersers", Proceedings of the 30th Annual ACM Symposium on Theory of Computing, pp 196-202, 1998