Short-lived signatures

Michael Colburn

A thesis in the

Concordia Institute for Information Systems Engineering

Presented in Partial Fulfilment of the Requirements for

the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

August 28, 2018

CONCORDIA UNIVERSITY

Division of Graduate Studies

This is to certify that the thesis prepared

By : **Michael Colburn**

Entitled : **Short-lived Signatures**

and submitted in partial fulfilment of the requirements for the degree of

Master of Applied Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee :

_____ Chair
Dr. Jun Yan

_____ CIISE Examiner
Dr. Amr Youssef

_____ External Examiner
Dr. Lata Narayanan

_____ Supervisor
Dr. Jeremy Clark

Approved by _____
Dr. Chadi Assi
Graduate Program Director

_____ 2018.

_____
Dr. Amir Asif
Dean of ENCS

ABSTRACT

Short-Lived Signatures

Michael Colburn

A short-lived signature is a digital signature with one distinguishing feature: with the passage of time, the validity of the signature dissipates to the point where valid signatures are no longer distinguishable from simulated forgeries (but the signing key remains secure and reusable). This dissipation happens "naturally" after signing a message and does not require further involvement from the signer, verifier, or a third party. This thesis introduces several constructions built from sigma protocols and proof of work algorithms and a framework by which to evaluate future constructions. We also describe some applications of short-lived signatures and proofs in the domains of secure messaging and voting.

Acknowledgments

First of all, I need to thank my supervisor Jeremy for taking a chance on me. Without his support and guidance this thesis wouldn't have been possible.

Many thanks go out as well to my friends near and far. With a special shout-out to Allyson Schmidt. Without our adventures I don't think I ever would have grown to love Montreal the way I do. And another to Kimberly Sayson for being my sounding board for practically everything. Thanks for putting up with me and keeping me in line.

Finally, I'm eternally grateful to my family and especially my parents, Heather and Glen, for their unconditional love and support even when I wasn't quite sure where my next steps might lead me. Thanks so much for everything. I miss you, Dad.

Contents

List of Tables

Chapter 1

Introductory Remarks

A digital signature is forever. Or at least, until the binding between key and identity is no longer reliable, or the underlying signature scheme is broken. This is often in contrast to what is strictly required in real world applications: a signature needs to only provide authenticity for a few seconds to conduct an authenticated key exchange or a few days for a signed email.

We might think this overreaching security does not matter. However, philosophically, it appears to violate the principle of least privilege if cryptographic proof that two entities once formed a secure connection or exchanged emails lives on for decades. Thus at best, such longevity is unnecessary. However in certain cases, we may explicitly not want this information to survive.

The idea that message authentication should not be universally verifiable is called deniability. Limiting who can verify messages could be done by limiting to a set of participants identified with public keys (designated verifier signatures), or by limiting in time (this work), or both. Achieving deniability could be an interactive process (OTR messaging) or non-interactive (this work).

Our intuition for what deniability means might be stronger than the actual guar-

antee. Informally, deniability means there is no cryptographic proof that Alice sent a particular message. There may still be circumstantial proof—logs or testimony—but there will be no proof beyond what would exist had Alice simply sent the message in plaintext with no message integrity.

A short-lived signature can be thought of as a proof of the following predicate: either Alice signed this message or someone did a lot of work recently. The work in the second clause can be done without Alice's knowledge or involvement. The amount of work can be considered to be d units of time, while recently can be considered to mean that the work started no earlier than a certain time t. While the signature is fresh, the difference between t and the present moment is smaller than d and thus only the first clause can be true. After the passage of time, t becomes greater than d time units ago and thus the second clause is potentially true. Losing the ability to distinguish the truth of the first clause from the second leads to our notion of deniability.

The fact that short-lived signatures provide deniability for the sender of a message without the sender needing to know the receivers private key, nor without having to interact with the sender, makes it uniquely qualified for achieving deniability in several practical scenarios, including sending signed email to a set of individuals who do not have known public keys. To our knowledge, ours is the first primitive to enable this.

Contributions. Our primary contributions are as follows.

1. We propose four constructions for short-lived signatures, some of which are compiled from a short-lived Σ-protocol.

2. We provide a framework for desirable properties for short-lived signatures, showing that our four constructions offer different subsets, and that the discovery of

an ideal signature that achieves all is still an open problem.

3. We provide some improved protocols that use this primitive, including a voting scheme.

Chapter 2

Preliminaries and Related Work

We consider in this section three types of related work: a set of building blocks we will use to construct short-lived signatures, primitives that could be used to achieve the same (or very similar) goals as a short-lived signature, and finally primitives that use the same building blocks for orthogonal goals. All of our protocols will be described using the discrete log setting but can be adapted to elliptic curves for faster computation and more compact representations.

2.1 Proof Systems

2.1.1 Proof Systems (without Zero-Knowledge)

The traditional notion of a proof system is a method by which a prover convinces a verifier of a claim. The prover provides evidence of their claim (this is called the witness) and the verifier in a either accepts or rejects the proof as appropriate.

There is a lot of variation amongst different styles of proof systems. One major distinction is whether the proof system is interactive or non-interactive. This basically describes whether the verifier needs to participate in the generation of the proof or can

simply accept a completed transcript of the proof itself to validate. In an interactive proof, the back and forth between prover and verifier is made up of a series of messages consisting of commitments, challenges and responses.

Unlike the classic definition of a proof, where an accepted proof is absolute evidence of the validity of a prover's claim, the proof systems we consider are probabilistic: with some negligible probability a prover may be able to construct a convincing proof for a false claim. The degree to which a proof system lowers this probability is called its *soundness*. For example, the soundness of the proofs we consider are overwhelming, meaning $1 - \epsilon$ where ϵ is a negligible probability in some security parameter[1]. By contrast, some proofs might have less soundness: for example, cut-and-choose protocols[EP84] might have soundness of $\frac{1}{n}$ for some value n which are highly, but not overwhelmingly, sound.

Another distinction is whether having unbounded computational abilities is of any assistance at all to the dishonest prover in having unsound proofs be accepted by an honest verifier. If computational abilities do not help, the proof is consider perfectly (or unconditionally) sound. If soundness requires the prover to be computationally bounded, it is called computationally sound (or an *argument* instead of a proof). We do not try here to categorize all proof systems but focus instead on a particular subset that is useful for cryptographic purposes called Sigma-Protocols (defined below). We will make extensive use of a technique called the Fiat-Shamir heuristic which results in a computationally sound proof. In this case, we assume a real-world entity cannot not generate enough fake proofs that one will happen to accept at random.

In order to prove the soundness of a proof system, a typical approach is to show that a dishonest verifier can extract the witness (or information equivalent to the knowledge being proven) from an honest prover producing valid proofs in polynomial

[1] If ℓ is the security parameter, $\mathsf{negl}(\ell) \leq \frac{1}{poly(\ell)}$.

time. Informally, the argument for why this works is as follows: if the witness can be extracted, then the proof must be 'aware' of the witness whereas an unsound proof would, by definition, not be aware of the witness. Of course, the prover may not want the verifier to extract the witness which motivates the idea of a zero-knowledge proof in the following section.

A proof system must also satisfy the property of completeness which essentially says that is capable of producing a validating proof when the witness is actually correct for the statement being proven. To have the completeness property, it should be the case that given an honest prover and verifier, the prover should convince the verifier of the validity of their claim with high or overwhelming probability.

2.1.2 Zero-Knowledge

A proof of knowledge on its own places no limits on what information the verifier learns by executing the protocol. For example, to prove that a particular number is composite, a prover could reveal the factorization of the number as part of their proof. However there are situations where it is highly desirable to limit the amount of information the verifier gains. On the extreme end, we can have zero knowledge proofs of knowledge which reveal nothing about the prover's claim aside from whether it is true or false. Zero knowledge is shown by the existence of a simulator which, given a claim, can produce an accepting transcript of the proof using only public information. Like soundness, the zero knowledge property can be either computational or perfect (*i.e.,* no amount of computational power would allow a verifier to learn any additional information).

2.2 Sigma Protocols

Proofs are define for a wide class of problems; indeed, a celebrated result shows that any statement in *PSPACE* can be proven with an interactive proof (and these can be made zero-knowledge under the assumption of a one-way function) [Sha92]. However these proofs are not necessarily, and often are not actually, efficient by any means. Therefore cryptographers have proposed zero-knowledge proof templates that are versatile for proving useful things but also efficient. The most well-studied cryptographic zero knowledge proof is likely the proof of knowledge of a discrete logarithm given by Schnorr[Sch91]. In fact, it is a special case of a general template of proof called a Sigma protocol (or sometimes informally, this class of proofs are called 'Schnorr proofs').

2.2.1 Interactive Sigma Protocols

Schnorr Proof of Knowledge of a Discrete Logarithm. Alice chooses secret key $x \in \mathbb{Z}_q$ for a prime-ordered multiplicative group of size q, computes public key $y = g^x \bmod p$ for large safe prime p (where $p = 2q+1$) and generator g of $\mathbb{G}_q$. Alice wants to prove to Bob that she knows x without telling Bob anything about x beyond the values of $\langle p, q, y \rangle$. Alice chooses random $a \in \mathbb{Z}_q$ and sends $b = g^a \bmod p$ to Bob. Bob chooses a random challenge c and sends it to Alice. Alice computes $d = a + cx \bmod q$ and sends it to Bob. Bob accepts the proof as correct if $g^d \stackrel{?}{=} b \cdot y^c \bmod p$.

This template of a proof can be generalized into a Sigma protocol. A Sigma protocol is a three move proof that takes its name from the shape of the flow of communication, which resembles the Greek capital letter sigma. In the first stage, the prover commits to a value and sends the commitment to the verifier. The verifier then replies with a challenge. The prover uses this challenge to compute a response

for the verifier. Finally, the verifier checks if the response is correct and accepts or rejects the proof accordingly.

We now illustrate some properties shown to be held by all sigma protocols; however we will illustrate them with the Schnorr ZKP specifically.

Public Coin. Since the verifier's challenge need only be a random value (and not of any particular structure or based on a secret that the verifier knows), we call it a public coin protocol. If we had a source for public random numbers, the verifier's role could be eliminated (we return to this later with the concept of beacons).

Completeness. One can verify that Schnorr is complete from the provided description: indeed $g^d = b \cdot y^c \bmod p$ holds when the values are constructed according the protocol using the actual x.

Special Soundness. Special soundness implies the existence of a polynomial time knowledge extractor that, given two accepting transcripts (a, c, r) and (a, c', r') where $c \neq c'$ and $r \neq r'$, allows recovery of the witness (or knowledge equivalent to it). In the case of a Schnorr Sigma protocol, the exact formula to recover the witness is $x = \frac{r-r'}{c-c'}$.

Transferability. Given an accepting proof transcript between a prover and a verifier, is the verifier able to also convince a third party to accept the proof? We call such a proof *transferable* if the verifier is able to do so (or non-transferable if not).

Honest Verifier Zero Knowledge. For the time being, we will make use of a weaker form of zero knowledge called *honest verifier zero knowledge* (HVZK) which assumes the verifier will always behave correctly and only achieves the zero knowledge

property in such a case. A dishonest verifier can make a transferable proof (*e.g.*,, using Fiat Shamir below).

2.2.2 Non-interactive Sigma Protocols

Sigma protocols are *interactive* three move protocols. For many applications it is useful to be able to convert these into non-interactive protocols. By using the *Fiat-Shamir heuristic*, we are able to make Sigma protocols non-interactive. To preserve the soundness property, the prover's commitment value must be fixed *before* the challenge is chosen so that a dishonest prover cannot construct a valid proof for a secret they do not actually have knowledge of. To ensure the challenge value is generated after the commitment is fixed, the Fiat-Shamir heuristic has the prover hash the commitment using a cryptographic hash function and use the result as the challenge value. For example, to make the Schnorr Sigma protocol non-interactive, the value of c is computed as $c = \mathsf{Hash}(b)$ and when verifying, the verifier must also check that the prover used the correct challenge (*i.e.,* check that $\mathsf{Hash}(b)$ is actually equal to c).

Weak vs. Strong Fiat-Shamir. There exist two variants of the Fiat-Shamir heuristic, dubbed weak and strong [BPW12]. In weak Fiat-Shamir, the challenge value is computed simply as the hash of the commitment value (*i.e.,* $c = \mathsf{Hash}(b)$ for a Schnorr proof). The strong variant of Fiat-Shamir also includes the statement to be proven in the challenge computation (*i.e.,* $c = \mathsf{Hash}(b, p, q, y)$ for a Schnorr proof, which is the commitment and the public key). Strong Fiat-Shamir is necessary in cases where the statement being proven can be adapted on the fly, so these values must also be committed to somehow. For our purposes, which rely on identification protocols, the weak version of Fiat-Shamir is sufficient to ensure the security of the

protocol.

Random Oracle Model. The Random Oracle Model is a way of modeling functions (usually hash functions) as if they were truly random to facilitate security arguments [BR93]. Given an input it has never seen before, the random oracle will generate a value at random (*e.g.,* from consecutive flips of a coin), record it and output the value. If it receives the same input value at a later time, it will notice it has seen this value before and return the same output value. It is possible to set the output value for a particular input in advance of calling the random oracle on that input value. This is called *programming* the random oracle. One paragraph. It basically models a hash function is a formal way. You give a query to the oracle (akin to the input to a hash) and it flips coins (for fixed lenght akin to output of hash), tells you the value, and writes it down so if you ask the same query again, it responds the same way. Random oracle in a proof can be programmed: choose the output first (randomly!) and then you can give it out and say when a certain query comes in, you will assign this random value to its output.

Completeness of Fiat-Shamir for Schnorr. To prove completeness, we need to show that the following equation holds: $g^d \stackrel{?}{=} b \cdot y^c \bmod p$. Recall that $b = g^a$ and $y = g^x$, so we have $g^d \stackrel{?}{=} g^a \cdot (g^x)^c \bmod p$. We can further simplify the right-hand side to $g^d \stackrel{?}{=} g^{a+cx} \bmod p$ and since we computed d as $d = a + xc$ in the final step of the protocol, we see that the original equation holds.

Soundness of Fiat-Shamir for Schnorr. To prove soundness, we assume an honest prover that will produce accepted transcripts using the witness x as specified by the protocol, and a malicious verifier called the extractor who will attempt to extra-filtrate the value of x from the transcript. If the extractor is successful, the

proof is indeed based on x and thus is sound (an unsound proof would be a transcript that accepts, is not based on x, yet in this case, it wouldn't be possible to extract x from it). At the same time, the extractor will use special powers real world adversary do not: being able to program the random oracle, thus an extractor will not work in real life when the random oracle is a real hash function. We will use the special soundness property and the fact that two accepting transcripts $\langle b, c, d\rangle$ and $\langle b, c', d'\rangle$ where $c \neq c'$ and $d \neq d'$ are sufficient to extract the witness x. The verifier lets the prover issue a first transcript $\langle b, c, d\rangle$, it then rewinds the prover to the point in time that it has chosen b but has not yet asked the random oracle for the hash of b to generate c and it instructs the oracle to erase its value of b. Then the oracle will pick a new random value c', overwhelmingly likely to be different than the first c and the prover will compute the corresponding d'. The extractor will use these values to compute x.

Zero-Knowledge of Fiat-Shamir for Schnorr. To prove zero-knowledge, we assume an honest verifier and a dishonest prover called the simulator. Because we are in the random oracle model, the simulator is allowed to program the random oracle. Its goal is to produce a transcript that is accepting to the honest verifier without actually knowing the witness. If it can do this, the transcript must be zero knowledge because it is possible to produce without knowing the witness. At the same time, because it will be instantiated in the real world with a real hash function that cannot be programmed, real provers have less power than the simulator and cannot use this to break soundness. The simulation strategy is as follows: the simulator chooses $d \in \mathbb{Z}_q$ at random, it asks the oracle for a random output value for not-yet-specified input and uses this as c, it computes $b = g^d \cdot y^{-c} \bmod p$, and finally it asks the oracle to program the output value it received as its response to the input b —

AND Composition (Chaum-Pedersen)

Proof

Input: public keys $\langle g, p, q, y_1 = g^x \rangle$, $\langle h, p, q, y_2 = h^x \rangle$ and signing key x.

1. Alice chooses $t \in_r \mathbb{Z}_p^*$ and uses it to compute $a = g^t$ and $b = h^t$.
2. She computes the challenge value $c = \mathsf{Hash}(a, b)$.
3. Finally, she calculates $r = t + cx$ and outputs $\sigma = \langle a, b, c, r \rangle$.

Verification

Input: public keys $\langle g, p, q, y_1 = g^x \rangle$, $\langle h, p, q, y_2 = h^x \rangle$ and transcript σ.

1. The verifier checks that $g^r = ay_1^c$ and $h^r = by_2^c$. If both of these equalities are true, then the proof is accepted.

Protocol 1: Non-interactive AND-Composed Proof of Equivalence of the Discrete Logarithm [CP92]

thus when the verifier asks the oracle, $\mathsf{Hash}(b) = c$.

2.2.3 AND-Composition of Sigma Protocols

Sigma protocols are fairly simple protocols but can be used as building blocks to construct ones which are more complex. Conjunction (logical *and*) is fairly straightforward. The prover executes two Σ-protocol in parallel with a common challenge. The verifier accepts only if they would accept both protocol runs individually.

It can be carried out in both interactive and non-interactive modes. We describe the non-interactive version in Protocol 1. To make it interactive, the c challenge value becomes a random value supplied by the verifier. This protocol is an example of an *and* composition of two Schnorr proofs with different public keys. It is also know as a Chaum-Pedersen proof of equivalence of the discrete log [CP92].

2.2.4 OR-Composition of Sigma Protocols

Disjunction (logical *or*), however, is more complicated.Trivially, this could be carried out as in conjunction where the verifier accepts if either of the protocol runs is

OR Composition (Cramer-Damgard-Schoenmakers)

Proof

Input: public key $\langle g, p, q, y_a = g^{w_a} \rangle$, signing key w_a and second public key $\langle g, p, q, y_b = g^{w_b} \rangle$.

1. Alice chooses $t_a \in_r \mathbb{Z}_p^*$ and uses it to compute $a_a = g^{t_a}$.
2. Then she simulates a proof of knowledge of w_b by choosing values $e_b, t_b \in_r \mathbb{Z}_p^*$ and computes a_b as $a_b = y^{-e_b} g^{t_b}$ and $r_b = a_b + e_b x$.
3. She computes the challenge value $c = \mathsf{Hash}(a_a, a_b)$ and uses it to also compute $e_a = c \oplus e_b$
4. Finally, she calculates $r_a = a_a + e_a x$ and outputs $\sigma = \langle a_a, a_b, r_a, r_b, e_a, e_b, c \rangle$.

Verification

Input: message m, public key $\langle g, p, q, y = g^x \rangle$, and signature σ.

1. The verifier checks that $g^{r_a} = a_a y_a^{e_a}$, $g^{r_b} = a_b y_b^{e_b}$ and that $c = e_a \oplus e_b$. If both of these equalities are true, then the proof is accepted.

Protocol 2: Non-interactive OR-Composed Schnorr Proof of Knowledge [CDS94]

true.The obvious drawback to this approach is that the verifier can now distinguish which of the two statements is true. By taking a slightly more involved approach, it is possible to structure a proof in such a way as to make it impossible for the verifier to distinguish which statement the prover was able to successfully prove.

The prover wishes to prove knowledge of one of two witnesses: w_a or w_b. We will assume the prover knows w_a. He will first simulate a proof for w_b to get a_b, e_b, r_b. He then computes a_a as usual and sends *both* commitment values a_a and a_b to the verifier. The prover receives random challenge c. However, this challenge is not used in this form directly. Using it directly in the proof of knowledge for w_a would leak which witness the prover has knowledge of since it has no interaction with the simulated proof yet. To connect it with the simulated proof, the prover combines c and e_b to get e_a (for example, by computing the $\oplus$ of the two values). The prover can now compute r_a according to the protocol and sends r_a, r_b, e_a, e_b. The verifier accepts if r_a and r_b equalities hold and $e_a \oplus e_b$ equals c. To make this proof non-interactive, the challenge is changed to be the hash of the two commitment values (*i.e.,* $c = \mathsf{Hash}(a_a, a_b)$). For an example using two non-interactive Schnorr

proofs of knowledge, see Protocol 2. Note that though Protocol 2 combines two of the same Sigma protocol, this type of composition also supports combining two different Sigma protocols. Another example of an OR composition is the protocol from Cramer, Damgård and Schoenmakers in [CDS94].

Though we lose the zero knowledge property through these types of composition, honest verifier zero knowledge is preserved. Note that we described disjunction and conjunction with only two witnesses, but since these properties can be composed with each other it is possible to perform these operations with an arbitrary number of witnesses.

2.2.5 PoWorK

A PoWorK is a proof construction introduced in [BKZZ16]. These proofs are indistinguishable proofs of work *or* knowledge (hence the name PoWorK). This means that upon seeing a PoWorK transcript, the verifier is unable to determine whether the prover actually had knowledge the witness, or instead carried out the necessary amount of work. The authors provide a framework for combining a Sigma protocol with a proof of work algorithm to achieve this goal. We will make use of one specific instantiation of a PoWorK as the basis of one of our short-lived constructions later on(see Figure 3.

2.3 Signatures

Digital signatures are a public key primitive used to bind an identity to a message. A signature scheme is composed of a pair of algorithms: $\mathsf{Sign}(x, y)$ that takes a message x, public key y and produces signature s, and $\mathsf{Verify}(m, s, z)$ which takes a message m, signature s and public key z and outputs accept if s is a valid signature

Proof of Work or Knowledge (PoWorK)

Proof of Knowledge Mode
Input: public key $\langle g, p, q, y = g^x \rangle$ and signing key x.

1. She selects a difficulty δ such that $\mathsf{TimeEst}(\delta)$ corresponds to the length of time that the proof should be valid. Let the output size of a collision-resistant hash function $\mathsf{Hash}()$ be λ bits.
2. She chooses $r \in_r \mathbb{Z}_p^*$ and computes $a = g^r$.
3. The challenge is constructed as follows: $c_0 = \mathsf{Hash}(a)$ and $c = c_0 \oplus c_1$, where $c_1 = (u, v, w)$ as defined next. $s \leftarrow \{0,1\}^\lambda$, $u = \mathsf{LSB}_\delta(\mathsf{Hash}(s, w))$, $v \leftarrow \{0,1\}^{\frac{\lambda}{2}-\delta}$, $w \leftarrow \{0,1\}^{\frac{\lambda}{2}}$.
4. She finally computes $d = a + cx$. She outputs signature $\sigma = \langle a, c, d, s, u, v, w, \delta \rangle$.

Proof of Work Mode
Input: public key $\langle g, p, q, y = g^x \rangle$.

1. At $\hat{t}_0$, Alice chooses $\hat{d} \in_r \mathbb{Z}_p$, and $\hat{c} \leftarrow \{0,1\}^\lambda$ and computes $\hat{a} = g^{\hat{d}} y^{\hat{c}}$, $\hat{c_0} = \mathsf{Hash}(\hat{a}, \hat{m}, b_{\hat{t}_0})$ and $\hat{c_1} = \hat{c} \oplus \hat{c_0}$.
2. She begins computing $\hat{s}$ such that $\hat{u} = \mathsf{LSB}_\delta(\mathsf{Hash}(\hat{s}, \hat{w}))$.
3. At $\hat{t}_1 \approx \hat{t}_0 + \mathsf{TimeEst}(\delta)$, Alice finds an acceptable $\hat{s}$. She outputs $\langle \hat{m}, \hat{\sigma}, \tau \rangle$ with signature $\hat{\sigma} = \left\langle \hat{a}, \hat{c}, \hat{d}, \hat{s}, \hat{u}, \hat{v}, \hat{w} \right\rangle$ and time information $\tau = \left\langle \hat{t}_0, b_{\hat{t}_0}, \delta \right\rangle$.

Verification
Input: message m, public key $\langle g, p, q, y \rangle$, and signature with timing information $\langle \sigma, \tau \rangle$.

1. At t_1, Bob checks that $t_1 < t_0 + \mathsf{TimeEst}(\delta)$.
2. He sets $c_0 = \mathsf{Hash}(a)$ and $c_1 = (u, v, w)$.
3. He checks that $c = c_0 \oplus c_1$ and $u = \mathsf{LSB}_\delta(\mathsf{Hash}(s, w))$.
4. He checks that $g^d = ay^c$. He outputs `accept` only if all checks hold.

Protocol 3: Non-interactive Proof of Work or Knowledge built from a Schnorr proof [BKZZ16]

of m under public key z, or otherwise it rejects the signature. For the purposes of this thesis we will focus on signatures based on *identification protocols* – interactive proofs of knowledge that prove possession of a private key to a verifier – however there are a variety of other methods of constructing digital signatures. The primary property that defines a digital signature scheme is unforgeability. That is, without prior knowledge of the private key, an attacker should not be able to create a signature that verifies. There are various degrees of unforgeability a protocol may possess:

under *universal forgery* an attacker would be able to create an accepting signature for any message, under *selective forgery* an attacker would be able to create an accepting signature only for select messages, and finally *existential forgery* where an attacker is able to create accepting signatures with no control the message which is signed. We can consider unforgeability in a variety of different attack scenarios that determine what other information the attacker has available: a *known message attack* where they have some number of messages – not of their choosing – and their accompanying accepting signatures, a *chosen message attack* allows the attacker to submit some list of messages to be signed and receive their accepting signatures, and finally an *adaptive chosen message attack* which is similar to a chosen message attack except the attacker is allowed to alter their submissions based on the outcome of prior results. Ideally a good digital signature scheme would be one that is existentially unforgeable even under an adaptive chosen message attack. In [PS96], the authors first show that the Schnorr signature scheme is secure against existential forgery in an adaptive chosen message attack. They go on to extend this more generally to any signature scheme made from an honest verifier zero knowledge identification protocol. The computational complexity of such an attack against these signature schemes would be equivalent to the underlying mathematically hard problem.

2.3.1 Schnorr Signatures

One of the simplest examples of a digital signature is a Schnorr signature. We can construct it by taking the Schnorr proof of knowledge, applying the Fiat-Shamir heuristic to it to make it non-interactive, and adding the message to the Fiat-Shamir hash.

ElGamal
Input: message m, public key $\langle g, p, y = g^x \rangle$ and signing key x.

1. Alice chooses random r, with $0 < r < p - 1$ and $\gcd(k, p-1) = 1$.
2. She computes $a = g^r \mod p$.
3. She uses the Fiat-Shamir heuristic to calculate the challenge $c = \mathsf{Hash}(m)$.
4. Finally, she computes $d = r^{-1}(c - xa) \mod (p-1)$.
5. She outputs the signature $\sigma = \langle m, d, a \rangle$.

Schnorr
Input: message m, public key $\langle g, p, q, y = g^x \rangle$ and signing key x.

1. Alice chooses random r, with $0 < r < q$.
2. She computes $a = g^r \mod p$.
3. She uses the Fiat-Shamir heuristic to calculate the challenge $c = \mathsf{Hash}(m, a)$.
4. Finally, she computes $d = a + xc \mod (p-1)$.
5. She outputs the signature $\sigma = \langle m, d, c \rangle$.

DSA
Input: message m, public key $\langle g, p, q, y = g^x \rangle$ and signing key x.

1. Alice chooses random r, with $0 < r < q$.
2. She computes $a = (g^r \mod p) \mod q$.
3. She uses the Fiat-Shamir heuristic to calculate the challenge $c = \mathsf{Hash}(m)$.
4. Finally, she computes $d = r^{-1}(c + xa) \mod q$.
5. She outputs the signature $\sigma = \langle m, d, a \rangle$.

Protocol 4: Signature generation for ElGamal, Schnorr and DSA constructions.

Schnorr Signature. Alice chooses secret signing key $x \in \mathbb{Z}_q$ and computes public verification key $y = g^x \bmod p$ for large safe prime p and generator g of $\mathbb{G}_q$. Alice signs message $m \in \{0,1\}^*$ by choosing random r and computing $s = \langle s_1, s_2, s_3 \rangle = \mathsf{Sign}_x(m, r) = \langle s_1 = g^r \bmod p, s_2 = \mathcal{H}(m \| s_1), s_3 = r + s_2 \cdot x \bmod q \rangle$. Bob verifies $\langle m, s \rangle$, the signature on the message, by ensuring $g^{s_3} \stackrel{?}{=} s_1 \cdot y^{s_2}$ and $s_2 \stackrel{?}{=} \mathcal{H}(m \| s_1)$.

2.3.2 DSA

DSA, the Digital Signature Algorithm, was specified as part of the Digital Signature Standard (DSS) by NIST in 1994. Along with ECDSA, its elliptic curve variant,

DSA is one of the most commonly used digital signature schemes. A DSA signature bears a strong resemblance to a Schnorr signature with one major change to the final response step borrowed from the ElGamal signature scheme (see Protocol 4).

2.3.3 Designative Verifier Signatures and Proofs

Designated verifier signatures. With a designated verifier signature, the following predicate is proven: either Alice signed this message or I know Bob's private key [JSI96]. Bob will believe the first clause since he knows Alice does not know his key, however he cannot convince anyone else that the first clause is true since he can make the second clause true. A designated verifier signature scheme can be built from a Σ-protocol with the same property. Short-lived signatures are compatible with designated verifier proofs/signatures and provide a different but complementary notion of deniability: one that is time-based instead of person-based. In an event when Alice knows Bob's private key, she can limit verification of the signature to Bob and for a period of time. When she does not know Bob's private key, she can only limit verification for a period of time.

A more minor practical concern is if Alice sends to multiple recipients, the size of a short-lived signature is constant in the number of receivers, while a designated verifier signature grows linearly in the number of receivers. For email, this likely matters little, but for constrained one-way messaging like SMS, it might play a role.

Relation to designated confirmer signatures. With a designated confirmer signature, signature verification is interactive rather than non-interactive. If the entity verifying the signature (the confirmer) (i) responds only to certain people or (ii) stops responding after a certain amount of time, the signature can no longer be verified. DV signatures can be thought of a non-interactive version of (i), while

short-lived signatures are a non-interactive version of (ii). Non-interaction simplifies the logistics of deniability considerably: Alice can send an email and go offline, and the signature will expire without any future involvement from her.

2.4 Other Building Blocks

Moderately-hard functions. Computing a moderately hard function consumes a certain amount of computational resources, which can be used to impose a price or time delay on an entity. These are variably called pricing [DN92], timing [FM97], delaying [GS98], or cost [GJMM98, Bac02] functions; and time-lock [RSW96, BN00, MMV11] or client [JB99, ANL00, DS01, WR03, WJHF04, DMR06, TBFG07, CMSW09, SKR+11] puzzles. Proof of work is sometimes used as an umbrella term [JJ99]. Among other applications, proof of work can be used to deter junk email [DN92, GJMM98] and denial of service attacks [JB99, DS01, Bac02, WR03, WJHF04], construct time-release encryption and commitments [RSW96, BN00], and mint coins in digital currencies [RS96, Bac02, Nak08].

We consider a puzzle as three functions: $\langle \mathsf{Gen}, \mathsf{Solve}, \mathsf{Verify} \rangle$. The generate function $p = \mathsf{Gen}(d, r, m)$ takes difficulty parameter d, randomness r, optionally a message string m and generates puzzle p. The solve function $s = \mathsf{Solve}(p)$ generates solution s from p. Solve is a moderately hard function to compute, where d provides an expectation on the number of CPU instructions or memory accesses needed to evaluate Solve. Finally, verification $\mathsf{Verify}(p, s)$ accepts iff s is a correct solution to p.

Time-stamping and carbon-dating. Time-stamping is a mechanism to establish a particular message is at least as old as a certain time. Time-stamping schemes use trusted entities [HS90, BdM91, BHS91, BdM93, BLLV98, PRQ+98, MB02] and have

been standardized.[2] Carbon dating [CE12, MMV11] is a trustless form of time-stamping that uses a moderately hard puzzle. Specifically, if difficulty d is a quantity of time, then publishing $\langle p = \mathsf{Gen}(d, r, m), s = \mathsf{Solve}(p)\rangle$ is proof that m is no newer than d units of time in the past.

Random beacons. While time-stamping proves a message is no newer than some past time, a random beacon [Rab83] provides a random number that is verifiably no older than some past time. Beacons can be based on an unpredictable source of randomness that is verifiable after the fact, such as financial data [WJHF04, CH10] or Bitcoin's blockchain [BCG15].

2.5 Other Related Work

The following are a few primitives and protocols that we do not make use of, however are related somewhat to our results. We thus provide them to distinguish how our results are different from them.

Deniable encryption. In deniable encryption schemes [CDNO97], message confidentiality is inherent. Further the sender must know the receivers' public key. With short-lived signatures, message confidentiality is an orthogonal concern: signatures can be applied to plaintext or ciphertext.

Time-lock encryption. In time-lock encryption [RSW96], an encrypted message becomes decryptable after an investment of a configurable amount of work. Short-lived signatures can be considered an analogue of time-lock encryption for signature schemes.

[2] ISO IEC 18014-3; IETF RFC 3161; ANSI ASC X9.95

Timed commitments and signatures. Timed commitments are an analogue of time-lock encryption for commitment schemes: a committed message becomes unhidden after an investment of a configurable amount of work [BN00]. The authors also extend their timed commitment construction to a 'timed signature' which likely sounds like the same primitive we are defining. However their timed signature solves a different problem relating to fair contract signing: a timed signature can be gradually released, and an investment of an amount of work can recover the signature if such a release is aborted early.

Chapter 3

Short-Lived Signatures and Proofs

A short-lived signature or proof is one whose truth erodes after a certain amount of time. By integrating a proof of work, our short-lived protocols allow indistinguishable forgeries to be created after an investment of time and resources. The motivation for this was provided in Chapter 1.

We begin with a 'folklore' short-lived signature which is to sign a message with a weak key. We improve this, combining existing primitives, to demonstrate that short-lived signatures are possible following this paradigm but require some additional things beyond just using a weak key. Later, we 'remix' the elements of this construction to build short-lived signatures in a more systematic way. Specifically, we build them by first building a short-lived zero knowledge proof and then using the Schnorr ZKP-to-Signature transformation to convert them into signatures. This is more modular than the enhanced folklore construction, which starts with a regular signature and ends up with a short-lived signature but cannot be used as a short-lived proof. Further both short-lived proofs and short-lived signatures are useful independently and we give example use cases in the next chapter that use one but not the other.

Folk+ construction steps	Improvement achieved
$\langle \mathsf{Sign}_{\bar{sk}}(m) \rangle$	Original construction with weak key $\bar{sk}$
$\langle \mathsf{Sign}_{sk}(m), \mathsf{TREnc}(sk) \rangle$	Strong sk; time-release encryption
$\langle (sk, pk) \leftarrow \mathsf{KeyGen}(b, n), \mathsf{Sign}_{sk}(m), \mathsf{TREnc}(n) \rangle$	Define earliest signing time
$\left\langle \mathsf{Sign}_{sk_l}(sk, b), \mathsf{Sign}_{sk}(m), \mathsf{TREnc}(sk) \right\rangle$	Plug into PKI where sk_l is a long-term signing key

Table 3.1: Progression from the original Folk construction to our enhanced short-lived Folk+ construction. Across the constructions, different signing keys are used: $\bar{sk}$ is a weak key, sk is full-strength key, and sk_l is a full-strength long-term key.

3.1 A folklore construction and an improvement

One folklore construction for short-lived signatures is often mentioned (although we cannot find an authoritative reference for it): sign the message with a weak key. By itself, this has numerous undesirable properties that we will sequentially explore, while also fixing them (a summary is given in Table 3.1). The result is an improved but impractical variant we call Folk+. Despite Folk+ being unwieldy, it serves as a pedagogical example: it shows the building blocks that can be used to construct a short-lived signature. In the next section, we start from scratch with these building blocks to make more concise short-lived signature schemes.

The notion of using a weak key is also presented in the analogous work on time-release encryption [RSW96] where the authors consider the idea of using a weak symmetric key. The authors (Rivest, Shamir, and Wagner) dismiss this idea for two reasons: exhaustive search is trivially parallelizable (a problem that we will revisit, as it is difficult to avoid in some of our constructions, and we denote constructions that achieve it as having puzzle sequentiality in Table 3.2) and the amount of work is configured on expected running time which could vary from actual running time in amounts that might make a practical difference. In the case of a signature scheme, this second issue is exacerbated by the fact that exhaustive search is not the best known

algorithm for finding signing keys in most signature schemes, and the best known algorithm could actually change based on the size of the key itself (*e.g.,* factoring or computing discrete logarithms). Thus determining an appropriate size for a weak key is non-trivial, not well-studied, and sensitive to the signature's setting and possibly its public parameters. A better approach is to use time-release encryption itself—a primitive designed for exactly what we want. The signer can sign the message with a full strength key sk (instead of a weak signing key $\bar{sk}$) and then time-release encrypt the full-strength key, $\mathsf{TREnc}(sk)$, with the desired difficulty.

If a recipient of a message signed in such a fashion sees this pair of values, a signature and a time-release encryption, $\langle \mathsf{Sign}_{sk}(m), \mathsf{TREnc}(sk) \rangle$, two necessary conditions exist for which this signature is a forgery: (i) the time-release encryption actually contains the signing key sk and (ii) the signature has been seen for a sufficient amount of time for sk to be released. As it stands, the first property (i) cannot be verified *a priori* — the value might the time-release encryption of any value. Thus in an improved version, we might want to achieve property that states that it should be apparent from inspection of the signature that forgeries can be created after an investment of resources. We call this property releasability in Table 3.2. Releasability may be possible to add to $\mathsf{Folk}+$ with an involved zero-knowledge proof but we do not pursue it here.

In terms of (ii), observe that there is no way to tell how long these values have been observed. If you receive an email signed in this fashion with a time release proportional to 1 day, it might be a valid message (*i.e.,* not a forgery) from Alice or it might be a just-completed forgery that Bob started creating yesterday. If we can prove the signature is freshly generated, this resolves this issue. We can do this using a beacon. Specifically we can incorporate a random value b from the beacon in the signing key (by using a random nonce n we define next). Note that it is not sufficient

to incorporate b into the signature itself, the message being signed, or the plaintext of the time-release encryption: since the signing key is time-released, once the key is known, the adversary can sign any message (using any beacon value) following any modified signature algorithm and re-encrypt it. The construction will appear as: $\langle (sk, pk) \leftarrow \mathsf{KeyGen}(b, n), \mathsf{Sign}_{sk}(m), \mathsf{TREnc}(n) \rangle$.

To complete this construction, we need a method for deriving a sk based on the beacon value. Since sk will be released upon completing the proof of work on the time-release encryption, it is not necessary that one can determine sk was derived from b from inspecting pk (otherwise, we'd require a zero knowledge proof or something like a verifiable random function). It will eventually become apparent, once sk is released, whether b was used. The simplest method to achieve this is to generate a random nonce n (of the same size as a signing key) and commit it with the beacon (*e.g.,* by hashing the two together and mapping into the keyspace) and use this combined value as the signing key $sk = \mathsf{Hash}(b, n)$. A verifier will repeat this process, once they have recovered n from the time-release encryption, by recomputing sk and pk using the asserted beacon value b and validating that these values actually produce the keys that were used. Note that if b is not integrated into the key in some way that is binding, then the beacon could be swapped out for a different value and this signature ceases to be short-lived since the accepting period of time cannot established.

The final consideration is that such a signature burns the signing key sk. While sk could be used a few times if messages were being sent from the same time interval, it will expire rapidly. This creates a PKI problem because signing keys need to be associated with identities through some mechanism, such as certificates. There is at least one simple solution: Alice can maintain a longterm signing key sk_l and certificate, and sign her ephemeral short-lived keys ($\mathsf{Sign}_{sk_l}(sk)$) to chain them back to her certificate. We consider this a necessary property of short-lived signatures,

which we call PKI compatibility in Table 3.2. This extra signature also gives us a place to include the beacon that will not become compromised after the time-release of the encryption as it is protected by the long-term key which is never released: $(\mathsf{Sign}_{sk_l}(sk, b))$. Thus b is no longer needed in $(sk, pk) \leftarrow \mathsf{KeyGen}(b)$ and the design becomes cleaner.

This completes our Folk+ construction. The first observation is that it is quite a long signature, involving two signatures and one encryption. Next, we will consider more concise constructions. Folk+ also lacks a few properties we will see in our next constructions, including the ability to generalize to a short-lived Σ-protocol and the ability to use arbitrary proof of work schemes instead of being bound to ones that admit time-release encryption. Finally, forgeries cannot be created until a valid signature is first created. Thus seeing an expired Folk+ short-lived signature proves that Alice signed at least one message after b was published; later constructions will enable forgeries apropos of nothing.

3.2 An initial attempt

Given the length of Folk+ (two signatures and one encryption), we now consider how we can 'remix' the elements of Folk+ (*e.g.,* a beacon and proof of work) into a more concise construction for a short-lived signature. To build a short-lived signature from scratch, we consider first building a short-lived Σ-protocol and then transforming it into a signature.

Recall for a proof of knowledge of some $x \in \mathcal{L}$, a Σ-protocol is a three move protocol where the prover sends a, receives c from the verifier, and responds with z. Applying the Fiat-Shamir heuristic ($c = \mathcal{H}(\mathcal{L}, a)$), it can be compiled into a non-interactive proof of knowledge. By binding the message to c ($c = \mathcal{H}(\mathcal{L}, a, m)$), it can

be transformed into a signature by the individual who knows x.

This attempt at a short-lived scheme is an adaptation of the idea for a 'pricing' or proof of work function proposed by Dwork and Naor[DN92]. They propose, as a proof of work, using a Fiat-Shamir signature scheme where the moderately hard problem is creating a forgery. To create a forgery, one can either find x or one can simulate fake signatures until the Fiat-Shamir value happens to equate. The degree to which the latter is possible is controlled by a security parameter t: the output of $c = \mathcal{H}(\mathcal{L}, a, m)$ is truncated to t bits.

In the scheme of Dwork and Naor, the fact that they use a signature instead a Σ-protocol as the basis for their proof of work is incidental. The message being signed plays no role in the Dwork-Naor construction so, with the hindsight of ensuring research on Σ-protocols, we can simplify their construction to create a short-lived Σ-protocol based on 'grinding' Fiat-Shamir (FS) values (hence the name of the next scheme, FS-Grind, which evolved out of this first attempt). We simply truncate c to t bits for a moderately sized value of t (*e.g.,* t=40 bits). Thus an accepting Σ-protocol transcript is due to either: the fact that it is a valid proof issued by a prover who knows x or it is a forged proof issued by someone that solved the proof of work. For now, there is no way to tell which is the case.

However if a verifier saw such a Σ-protocol transcript and knew it was 'freshly' generated—the elapsed time is less than the time required to solve the proof of work—he could conclude it is a valid proof issued by the prover who knows x. As the elapsed time grows, he can no longer conclude this and the proof's validity becomes uncertain (or in the signature case, the signature becomes deniable). The freshness can be accomplished by incorporating a beacon value b, however we will discuss one point further before specifying where to insert b.

The final step will be to transform this Σ-protocol into a signature. For a Fiat-

Shamir-based Σ-protocol, the transformation is to add the message to the hash used to compute the challenge c. In our case, the truncation of c to allow grinding a forged proof also impacts the collision resistance of the hash; specifically, the hash is no longer collision resistant. This is a problem because collision resistance must hold for both the message m and the beacon b (assuming b is added to this hash) — therefore the transformation does not work in this, and it does not work for either the signature scheme or the Σ-protocol. If the beacon can be integrated in a binding way to a value other than c in the proof, this construction could possibly be repaired to work for Σ-protocols, however we will fix it in a more thorough way that works for both signatures and Σ-protocols in the next section.

3.3 FS-Grind

The main component of an FS-Grind signature is a modified Chaum-Pedersen protocol which is a proof of knowledge of the equality of two discrete logarithms (see Section 2.2.3). The base of the first discrete logarithm is the public key, g, while the second base is a hash h of the message and a random beacon value. The proof will prove knowledge of x given $\langle g, h, g^x, h^x \rangle$. This allows us to incorporate b (and in the case of a signature, both b and m) into the proof transcript with a full-fledged collision-resistant hash function, repairing the issue with our previous attempt. Note that for every proof (and signature), the value of h changes from signature to signature but the value of g and g^x is always the same—since g^x is the signer's public key, this means the signer can use the same key for multiple signatures which is important for using an established PKI.

FS-Grind is given in Protocol 5. We specify the signature version (for a short-lived Σ-protocol, m is dropped and the terminology shifts). The challenge value is a

FS-Grind

Signature
Input: message m, public key $\langle g, p, q, y\rangle$ and signing key x.

1. At t_0, Alice obtains randomness $b_{t_0} = \mathsf{GenRand}()$ from beacon.
2. She selects a difficulty δ such that $\mathsf{TimeEst}(\delta)$ corresponds to the length of time that the signature should be accepted.
3. She computes $h = \mathsf{Hash}(m, b_{t_0})$, sets $y_1 = y$ and $y_2 = h^x$.
4. She chooses $r \in_r \mathbb{Z}_p^*$, computes $a_1 = g^r$, $a_2 = h^r$, $c = \mathsf{Trunc}_\delta(\mathsf{Hash}(a_1, a_2))$ and computes $d = r + cx$. She outputs $\langle m, \sigma, \tau\rangle$ where signature $\sigma = \langle h, y_1, y_2, a_1, a_2, c, d\rangle$ and time information $\tau = \langle t_0, b_{t_0}, \delta\rangle$.

Verification
Input: message m, public key $\langle g, p, q, y\rangle$, and signature $\langle \sigma, \tau\rangle$.

1. At t_1, Bob checks that $g^d = y_1^c a_1$, $h^d = y_2^c a_2$ and $c = \mathsf{Trunc}_\delta(\mathsf{Hash}(a_1, a_2))$. He then checks that $\mathsf{VerifyPuz}(\mathsf{GenPuz}(b, \delta), s_b)$. Finally he checks that $t_1 < t_0 + \mathsf{TimeEst}(\delta)$. He outputs `accept` only if all checks hold.

Forgery
Input: a forged message $\hat{m}$.

1. Selective Forgery: At $\hat{t}_0$, Eve computes $\hat{h} = \mathsf{Hash}(\hat{m}, b_{\hat{t}_0})$, chooses $y_2 \in_r \mathbb{Z}_p$, $\hat{d} \in_r \mathbb{Z}_p$ and $\hat{c} \in_r \mathbb{Z}_p$. She computes $\hat{a}_1 = y_1^{-\hat{c}} g^{\hat{d}}$ and $\hat{a}_2 = y_2^{-\hat{c}} \hat{h}^{\hat{d}}$. She selects difficulty $\hat{\delta}$ and checks if $\hat{c} = \mathsf{Trunc}_{\hat{\delta}}(\mathsf{Hash}(\hat{a}_1, \hat{a}_2))$. If this does not hold, she selects new random values and recomputes. She will expect to repeat this for $\mathsf{TimeEst}(\delta)$.
2. Completion: At $\hat{t}_1 \approx \hat{t}_0 + \mathsf{TimeEst}(\delta)$, Alice finds an acceptable $\hat{c}$. She outputs $\langle \hat{m}, \hat{\sigma}, \tau\rangle$ with signature $\hat{\sigma} = \left\langle \hat{h}, y_1, y_2, \hat{a}_1, \hat{a}_2, \hat{c}, \hat{d}\right\rangle$ and time information $\tau = \langle \hat{t}_0, b_{\hat{t}_0}, \delta\rangle$.

Protocol 5: Signature and verification for FS-Grind instantiated with Schnorr signature.

hash of the two commitment values, one using each base exponentiated to the same randomness, the base which contains the hash of the message as well as that base exponentiated with the signing key. Before use, the challenge is truncated to the desired difficulty level which determines the validity period of the signature. The final response is the same as a standard Schnorr signature.

The verification step of an FS-Grind signature also works much the same as in a Schnorr signature except that the check has to be done against both bases. In addition, the verifier should check that the correct challenge value was used.

To forge an FS-Grind signature, Eve selects a random challenge and response value and uses these to calculate commitment values. She then checks if the random challenge value matches the expected value from hashing the commitments. If not, she can pick new random values and try again.

3.4 Workflow

The Workflow construction is quite similar to FS-Grind. They both admit forgeries upon finding a certain hash output (where the size depends on a chosen difficulty parameter). This hash value is used directly or indirectly as the challenge value in a standard Σ-protocol, and thus both FS-Grind and Workflow can be used as short-lived Σ-protocols or transformed into signatures using the standard transformation. Both grapple with the issue that a full-size hash function is needed for security, while finding a "correct" a truncation of the hash to δ bits enables a proof of work that can be easily integrated into the protocol.

In FS-Grind, we move the message and beacon out of the challenge and into two new values, linked by their discrete logarithm being the secret key, base the protocol on the AND-composition within an Σ-protocol, and have the challenge be small. With Workflow, we keep the message and beacon in the challenge but work on structuring the challenge to be both (1) full length for collision resistance, and (2) accepting even if only some subset of the challenge are 'correct.' We base Workflow on recent work by Baldimtsi *et al.* [BKZZ16] on building indistinguishable proofs (or Σ-protocols) of work or knowledge (which they call PoWorKs). PoWorKs and Workflow are very similar with the following differences: Workflow is a signature, Workflow conforms to being short-lived while PoWorKs are designed with a different application in mind (see Section 2.2.5), and to realize the short-lived property, Workflow uses a beacon

value.

The intuition of Workflow is as follows: for Σ-protocols in general, if the proof transcript is a forgery provided by a prover that does not know the witness, there is generally only a single value from a large message space that will make the proof accept. The job of the malicious prover is to find this value. Typically it cannot be found directly, it can only be found through trial-and-error — in this case, it is because the value is a preimage to a preimage resistant hash function. With Workflow, the proof is set up as an OR (see Section 2.2.4): there are two sub-challenges that are combined into a single challenge, and once one sub-challenge is programmed by the adversary, the other sub-challenge has a fixed value that the prover needs to deal with. To transform the proof into a signature, we use the same standard transformation (Fiat-Shamir to make it interactive, and adding the message to the hash to make it a signature).

In the proof of knowledge setting (*i.e.,* when the prover actually knows the signing key), the prover will generate a puzzle solution first and work backwards to derive the corresponding puzzle. The puzzle is defined by the value of the work-side sub-challenge, combined with the knowledge-side sub-challenge, and combined to form the actual challenge in the proof/signature. Alternatively, in the proof of work setting (*i.e.,* when the signer is forging a signature), the knowledge-side sub-challenge is chosen first and the commitment value is calculated from it. The puzzle-side sub-challenge can now be calculated with both challenge values fixed, and the prover begins to solve the proof of work to find the one value that will accept. The one value that will accept is one value from a set of values defined by how many bits the hash is truncated to.

The signature has been modified to be forgeable but it is not yet a short-lived signature since it lacks any timing information to allow for expiration. As a final

Workflow

Signature
Input: message m, public key $\langle g, p, q, y\rangle$ and signing key x.

1. At t_0, Alice obtains randomness $b_{t_0} = \mathsf{GenRand}()$ from a beacon.
2. She selects a difficulty δ such that $\mathsf{TimeEst}(\delta)$ corresponds to the length of time that the signature should be valid. Let the output size of a collision-resistant hash function $\mathsf{Hash}()$ be λ bits.
3. She chooses $r \in_r \mathbb{Z}_p^*$ and computes $a = g^r$.
4. The challenge is constructed as follows: $c_0 = \mathsf{Hash}(a, m, b_{t_0})$ and $c = c_0 \oplus c_1$, where $c_1 = (u, v, w)$ as defined next. $s \leftarrow \{0,1\}^\lambda$, $u = \mathsf{LSB}_\delta(\mathsf{Hash}(s, w))$, $v \leftarrow \{0,1\}^{\frac{\lambda}{2}-\delta}$, $w \leftarrow \{0,1\}^{\frac{\lambda}{2}}$.
5. She finally computes $d = a + cx$. She outputs $\langle m, \sigma, \tau\rangle$ where signature $\sigma = \langle a, c, d, s, u, v, w\rangle$ and time information $\tau = \langle t_0, b_{t_0}, \delta\rangle$.

Verification
Input: message m, public key $\langle g, p, q, y\rangle$, and signature with timing information $\langle \sigma, \tau\rangle$.

1. At t_1, Bob checks that $t_1 < t_0 + \mathsf{TimeEst}(\delta)$.
2. He sets $c_0 = \mathsf{Hash}(a, m, b_{t_0})$ and $c_1 = (u, v, w)$.
3. He checks that $c = c_0 \oplus c_1$ and $u = \mathsf{LSB}_\delta(\mathsf{Hash}(s, w))$.
4. He checks that $g^d = ay^c$. He outputs `accept` only if all checks hold.

Forgery
Input: message $\hat{m}$ for forged signature.

1. Selective Forgery:
   At $\hat{t}_0$, Eve chooses $\hat{d} \in_r \mathbb{Z}_p$, and $\hat{c} \leftarrow \{0,1\}^\lambda$ and computes $\hat{a} = g^{\hat{d}} y^{\hat{c}}$, $\hat{c_0} = \mathsf{Hash}(\hat{a}, \hat{m}, b_{\hat{t}_0})$ and $\hat{c_1} = \hat{c} \oplus \hat{c_0}$. She begins computing $\hat{s}$ such that $\hat{u} = \mathsf{LSB}_\delta(\mathsf{Hash}(\hat{s}, \hat{w}))$.
2. Completion: At $\hat{t}_1 \approx \hat{t}_0 + \mathsf{TimeEst}(\delta)$, Eve finds an acceptable $\hat{s}$. She outputs $\langle \hat{m}, \hat{\sigma}, \tau\rangle$ with signature $\hat{\sigma} = \left\langle \hat{a}, \hat{c}, \hat{d}, \hat{s}, \hat{u}, \hat{v}, \hat{w}\right\rangle$ and time information $\tau = \left\langle \hat{t}_0, b_{\hat{t}_0}, \delta\right\rangle$.

Protocol 6: Signature and verification for Workflow instantiated with Schnorr signature.

modification, we return to the challenge hash value and add a random beacon value to the computation from the time the signature is being generated (or any time earlier). This value will show that the signature could have been generated no earlier than the time corresponding to the included beacon value. Combined with the time requirement to solve the proof of work, this fixes a time after which this signature will become indistinguishable from one which has been forged. For a detailed description

of Workflow signatures, see Protocol 6.

Finally, we identify and fix a small flaw in the original PoWorK proposal. In Protocol 6, what we call v is chosen at random regardless of whether the signature is a forgery or not. In the original paper [BKZZ16], this value v is chosen to be an extension of the hashed bits comprising u when the PoWorK is knowledge-based and it ends up (with overwhelming probability) being random when the PoWorK is work-based. Thus knowledge/work is not truly indistinguishable.[1]

3.5 Carbon

Like our previous constructions, Carbon begins with a Schnorr Σ-protocol as a base. However unlike in our earlier constructions, we will integrate the proof of work into the commitment phase instead of the challenge phase of the protocol. This allows for a greater degree of flexibility in choosing a proof of work protocol.

To generate a Carbon signature in the proof of knowledge setting, Alice will first commit to a random value and use it to generate the proof of work puzzle. Upon solving this proof of work, Alice can continue with generating the signature. She will compute a hash of the message and a beacon value to form the challenge and fix the beginning of the validity period of the signature. Lastly, she will calculate the response value and output the signature information.

To forge a Carbon signature, Alice computes the challenge value as before and generates a random response value. She uses these values to calculate what the commitment should be so that she is able to generate the proof of work puzzle. Upon

[1]The exact issue arrises in the security proof of statistical indistinguishability within Theorem 2 of [BKZZ16]. The authors make the following statement: "it is obvious that $\mathsf{Solve}(1^\lambda, h, \mathsf{puz}))$ outputs a random soln from the solution set of puz, which is identically distributed to the solution soln in $(\mathsf{puz}, \mathsf{soln}) \leftarrow \mathsf{SampleSol}(1^\lambda, h, \mathsf{puz}))$." However the authors do not prove they are identically distributed, and indeed they are not.

Carbon

Signature
Input: message m, public key $\langle g, p, q, y\rangle$ and signing key x.

1. At t_0, Alice chooses $a \in_r \mathbb{Z}_p^*$ and computes $b = g^a$. Alice generates puzzle $p_b = \mathsf{GenPuz}(b, \delta)$ with difficulty δ. She begins computing $s_b = \mathsf{Solve}(p_b)$.
2. At $t_1 \gg t_0$, Alice completes computation of s_b.
3. At $t_2 \geq t_1$, Alice obtains randomness $r_{t_2} = \mathsf{GenRand}()$ from beacon. She generates $c = \mathsf{Hash}(r_{t_2}, m)$ and computes $d = a - cx$. She outputs $\langle m, \sigma, \tau\rangle$ where signature $\sigma = \langle b, c, d, s_b\rangle$ and time information $\tau = \langle t_2, \delta\rangle$.

Verification
Input: message m, public key $\langle g, p, q, y\rangle$, and signature with timing information $\langle \sigma, \tau\rangle$.

1. At t_3, Bob checks that $b = y^c g^d$. He checks $r_{t_2} = \mathsf{GenRand}(t_2)$ and checks that $c = \mathsf{Hash}(r_{t_2}, m)$. He then checks that $\mathsf{VerifyPuz}(\mathsf{GenPuz}(b, \delta), s_b)$. Finally he checks that $t_3 < t_2 + \mathsf{TimeEst}(\delta)$. He outputs `accept` only if all checks hold.

Forgery

input: timing information τ and forged message $\hat{m}$

1. Selective Forgery: At $\hat{t}_0$, Eve chooses $\hat{d} \in_r \mathbb{Z}_p$ and generates $\hat{c} = \mathsf{Hash}(r_{t_2}, \hat{m})$. She computes $\hat{b} = y^{\hat{c}} g^{\hat{d}}$. Eve generates puzzle $\hat{p_b} = \mathsf{GenPuz}(\hat{b}, \delta)$ with difficulty δ. She begins computing $\hat{s_b} = \mathsf{Solve}(\hat{p_b})$.
2. Completion: At $\hat{t}_1 \gg t_0$, Eve completes computation of $\hat{s_b}$. She outputs $\langle \hat{m}, \hat{\sigma}, \tau\rangle$ with signature $\hat{\sigma} = \left\langle \hat{b}, \hat{c}, \hat{d}, \hat{s_b} \right\rangle$ and *same* time information $\tau = \langle t_2, \delta\rangle$.

Protocol 7: Signature, verification and forgery of Carbon.

completion of the puzzle, Alice can output an accepting but forged Carbon signature. For a full description of both of the signature modes and verification, see Protocol 7.

Carbon features two main drawbacks. The first is that upon witnessing a fresh, valid signature, Bob may be able to prevent the signature from expiring naturally by continuing the proof of work (the others do not have this drawback and schemes that do not are considered *Persistent* in our comparison below).

The second drawback is that in order to generate a valid signature both the legitimate prover and the forger have to carry out the full proof of work (while the verifier does not). This proof of work can be pre-computed (it does not depend on the signing key or message) and multiple messages might be batched together with

one-precomputed puzzle, however this is a significant drawback. In our comparison below, we say Carbon does not achieve *No pre-computation* as a result. The trade-off is *PoW-independence* where Carbon can use any proof of work scheme that provides the carbon-dating property.

3.6 Security

The protocols Carbon, FS-Grind and Workflow are very similar to each other. We outline some differences below. However to prove they are proper signatures, we use FS-Grind as a representative on the family of protocols. We first prove it a short-lived Fiat-Shamir-based Σ-protocol and then we rely on the standard Σ-protocol-to-signature transformation for constructing a signature (we do not prove this transformation and instead refer the reader to [Sch91]).

Completeness. To prove completeness, we need to show that the following two equations hold: $g^d \stackrel{?}{=} y_1^c a_1 \bmod p$ and $h^d \stackrel{?}{=} y_2^c a_2 \bmod p$. Beginning with g^d, recall that $a_1 = g^r$ and $y_1 = g^x$, so we have $g^d \stackrel{?}{=} g^r \cdot (g^x)^c \bmod p$. We can further simplify the right-hand side to $g^d \stackrel{?}{=} g^{r+cx} \bmod p$ and since we computed d as $d = r + cx$ in the final step of the protocol, we see that the original equation holds. Now for h^d, recall that $a_2 = h^r$ and $y_2 = h^x$, so we have $g^d \stackrel{?}{=} g^r \cdot (h^x)^c \bmod p$. We can further simplify the right-hand side to $h^d \stackrel{?}{=} h^{r+cx} \bmod p$ and since we computed d as $d = r + cx$ in the final step of the protocol, we see that the original equation holds.

Short-lived Special Soundness. To prove soundness, we assume an honest prover that will produce accepted transcripts using the witness x as specified by the protocol, and a malicious verifier called the *extractor* who will attempt to exfiltrate the value of x from the transcript. If the extractor is successful, the proof is indeed based on x and

thus is sound (an unsound proof would be a transcript that accepts, is not based on x, yet in this case, it wouldn't be possible to extract x from it). At the same time, the extractor will use special powers real world adversaries do not: being able to program the random oracle, thus an extractor will not work in real life when the random oracle is replaced by a real hash function. We will use the special soundness property and the fact that two accepting transcripts $\langle a_1, a_2, c, \rangle$ and $\langle a_1, a_2, c', r' \rangle$ where $c \neq c'$ and $r \neq r'$ are sufficient to extract the witness x. The verifier lets the prover issue a first transcript $\langle a_1, a_2, c, \rangle$, it then rewinds the prover to the point in time that it has chosen a_1 and a_2 but has not yet asked the random oracle for the hash of a_1 and a_2 to generate c and it instructs the oracle to erase its value of a_1 and a_2. Then the oracle will pick a new random value c', overwhelmingly likely to be different than the first c and the prover will compute the corresponding r'. The extractor will use these values to compute x as $x = \frac{s-s'}{c-c'}$.

Non-interactive Zero-Knowledge. To prove zero-knowledge, we assume an honest verifier and a dishonest prover called the simulator. Because we are in the random oracle model, the simulator is allowed to program the random oracle. Its goal is to produce a transcript that is accepting to the honest verifier without actually knowing the witness. If it can do this, the transcript must be zero knowledge because it is possible to produce without knowledge of the witness. At the same time, because it will be instantiated in the real world with a real hash function that cannot be programmed, real provers have less power than the simulator and cannot use this to break soundness. The simulation strategy is as follows: the simulator gets a value b_{t_0} from the beacon, computes $h = \mathsf{Hash}(b_{t_0})$ and chooses $y_2, d \in \mathbb{Z}_p^*$ at random, it asks the oracle for a random output value for not-yet-specified input and uses this as c, it computes $a_1 = g^d \cdot y_1^{-c} \bmod p$ and $a_2 = h^d \cdot y_2^{-c}$, and finally it asks the oracle

	Setting independent	PKI compatibility	Allows Σ-protocol	PoW-independent	No pre-computation	No persistence	Releasability	Key-only forgeries	Puzzle sequentiality	Modular exponentiations
Folk	•				•	•				1
Folk+		•		○	•	•			•	3
FS-Grind	•	•	•		•	•	•	•		2
Workflow	•	•	•	○	•	•	•	•		1
Carbon	•	•	•	•			•	•		1

Table 3.2: A comparison of various short-lived signature constructions.

to program the output value it received as its response to the input (a_1, a_2) — thus when the verifier queries the oracle, $c = \mathsf{Hash}(a_1, a_2)$.

3.7 Evaluation

No short-lived scheme we have described possesses all of the properties listed here and they each offer a different subset allowing them to be useful in a variety of situations. This list is not exhaustive and there may be other useful properties we have overlooked in our evaluation.

Setting independent. Constructions achieving this property are those which do not require a specific mathematical setting (*e.g.,* discrete log, RSA) in order to function. This allows our constructions to adapt to future advances in cryptographic attacks against digital signatures.

All of our protocols were described generically (in the case of Carbon, Folk) or using the discrete log setting for the sake of consistence and readability (FS-Grind, and Workflow). The discrete log constructions can be adapted to other settings, such

as using elliptic curves, as necessary. Though Folk+ was also described generically, in order to satisfy the puzzle sequentiality property as well it must be in the RSA setting.

PKI compatibility. Most uses of digital signatures today rely on the TLS public key infrastructure or the PGP web of trust. Ideally any short-lived signature constructions should be compatible with these (*i.e.,* they make use of long term signing keys).

All of the short-lived signature schemes we describe, with the exception of the basic Folk construction, allow for these long term keys to be used for signing.

Allows Σ-protocol. We defined our short-lived constructions as signatures, however we have identified use cases which would rely on short-lived *proofs* instead. It would be preferable if our constructions support both of these modes, signature and proof.

Neither of the folklore constructions (Folk, Folk+) are capable of being used in a proof mode. FS-Grind, Workflow and Carbon can be used in a proof mode, however, as they are built from Sigma protocols. To do so, the only necessary change is to omit the message m from the protocol (and for interactive proofs, the challenge value may be provided by the verifying party).

PoW-independent. Constructions with this property do not prescribe a particular proof of work algorithm and will work with a variety of them.

Folk requires exhaustive search on a key. Folk+, meanwhile, can use any time-release encryption scheme. However, we only assign ∘to Folk+ due to its unique ability to satisfy the puzzle sequentiality property. In order to achieve puzzle sequentiality, the proof of work algorithm must be carefully chosen. A Carbon signature can be

made using any non-interactive proof of work. FS-Grind specifies grinding a specific hash or hash image prefix. Workflow is described using a similar proof of work to FS-Grind, however a generic definition is also given by [BKZZ16] that will work for a short-lived signature as well.

No pre-computation. To what degree, if any, is pre-computation required for a particular short-lived scheme. Constructions with this property allow for faster and more frequent signature generation.

With the exception of Carbon, none of the other short-lived signatures we described require any computation to be carried out in advance.

No persistence. Since our short-lived constructions integrate a proof of work into a signature (or proof), it may be possible to extend the lifetime of a signature slightly beyond what was intended by continuing the proof of work. We call this persistence and consider it undesirable.

None of Folk, Folk+, FS-Grind or Workflow allow for signature persistence. Carbon is the only one of our constructions missing this property. Upon witnessing an accepting Carbon short-lived signature, the verifier can prevent it from expiring by continuing the proof of work. However, once even a Carbon signature expires it cannot be made to appear fresh again.

Releasability. Consider a short-lived signature σ on message m as $\langle m, \sigma, \Delta t, b \rangle$ where Δt is the time (or work) required to create forgeries and b is the beacon value or freshness of the signature. If a short-lived signature is releasable ($\bullet$), it should be clear from the inspection σ itself that the asserted values of Δt and b are the values actually used in the construction of σ. If the fact that one or both of the asserted values Δt and b are known to be used only after completing a proof of work, it is

said not to be releasable (but it is still a short-lived signature). If the assertions can never be established, it is not a short-lived signature.

In the case of Folk+, there is no guarantee that the time-released signing key will actually produce valid signatures when used until after the work has been completed. All of the other proposed short-lived schemes do not make use of signing keys in order to execute forgeries, relying solely on the proof of work instead so seeing one of these signatures assures releasability.

Key-only forgeries. How much does the adversary have to see before she can generate a forgery? By allowing forgeries without a valid signature having ever been created, the signer is afforded a slightly greater degree of deniability.

For Folk+, a public key and at least one valid signature is necessary. In addition to a valid signature, Folk also requires the release of the private signing key. Our other constructions, FS-Grind, Workflow and Carbon, allow forgeries with just knowledge of the public key.

Puzzle sequentiality. Short-lived signatures blend together the human and computational notions of time. Inherently sequential proof of work algorithms are those for which future steps of the puzzle rely on past intermediate results. This allows for better bounds on time estimates as it will limit parallelizing the computation of proofs of work.

We only know of sequential puzzles for time-release encryption, and thus Folk+.

Modular exponentiations To compare the complexity of each of the short-lived constructions, we can consider the number of modular exponentiations required to generate a fresh signature (rather than a forgery, which is an inherently expensive process) as this is likely to be the mostly computationally intensive step of each

construction.

Folk only requires a signature with a weak key, requiring one modular exponentiation. Folk+ requires one for each of the two signatures, as well as one more for the time-release encryption for a total of three exponentiations. FS-Grind, which is built from the parallel composition of two signatures, requires two exponentiations. Workflow and Carbon both each only require a single modular exponentiation.

3.8 Extensions

Fine-grained difficulty. In FS-Grind and Workflow, the difficulty of the puzzle consists of finding a pre-image that produces a specific hash output that is truncated to δ bits. The probability of a hash output matching is thus $1/2^{\delta}$. An increase or decrease in δ by 1 will double or halve the difficulty. We would like finer-grain control where the difficulty can be set to an exact amount. We illustrate a transformation on Workflow (and the same principle can be applied to FS-Grind).

The crux of Workflow is computing the value $\mathsf{LSB}_{\delta}(\mathsf{Hash}(s, w))$ such that it equals a given value u. To allow finer grained control, we were permit hash outputs that are both exactly u and also close to u as defined by some distance. For example, $\mathsf{LSB}_{\delta}(\mathsf{Hash}(s, w))$ should be in the interval $[u, u + \gamma]$ (technically $u + \gamma \bmod \delta$). In this case, the probability is $\gamma/2^{\delta}$ which can take on any probability given a choice of both γ and δ. To preserve indistinguishability, when an honest signer is producing an accepting signature, they will compute $u = \mathsf{LSB}_{\delta}(\mathsf{Hash}(s, w))$ and instead of outputting u itself, they will construct the interval $[u, u + \gamma]$ and choose a uniformly random value in this interval as u.

Designated verifier. In addition to regular digital signatures, short-lived signatures can also compliment designated verifier (DV) signatures in some situations. DV

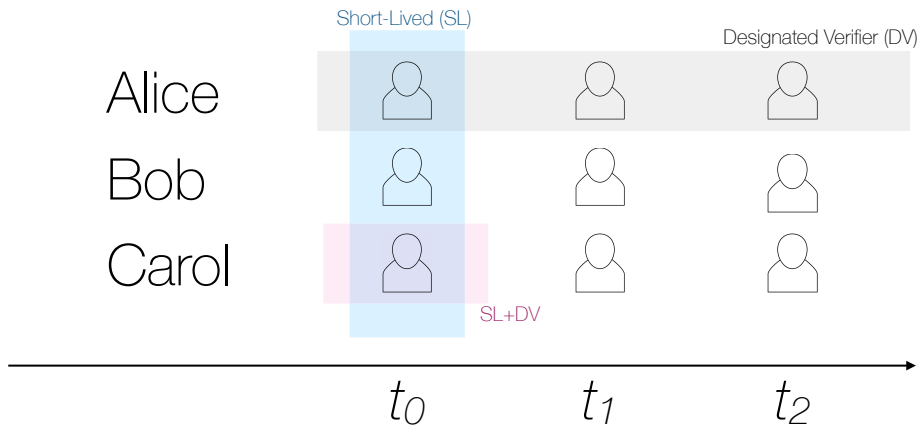

Figure 3.1: Comparing the properties of short-lived signatures with designated verifier signatures. Specifically, which parties are able to verify a signature over time.

signatures limit verification to a subset of people while short-lived signatures allow anyone to verify within a specific timeframe.

A short-lived signature has no problems scaling to an arbitrary number of verifiers, while a DV signature size will increase as the set of verifiers grows, making these signatures unwieldy for groups larger than a few participants. DV signatures also require the verifier to have a public key and for the signer to be aware of it at the time of signature generation. Whereas for our short-lived signatures the signer does not need to know who will be verifying the signatures, let alone their public keys.

Though we contrast short-lived signatures with designated verifier signatures here, they do serve different purposes and it would not be simple or make sense to replace a DV signature with a short-lived signature in certain cases. In such a scenario they could actually be combined to produce a short-lived designated verifier signature which would limit both the scope of verifiers as well as the timeframe in which they are able to undeniably verify the signature (see Figure 3.1).

Estimating time. A short-lived signature could find use in practically any situation a conventional digital signature is used. The shorter the required validity period, the better. This is due to how we define time in our protocols. There is no obvious way to integrate real world time so we use expected computation time as an analogue. In practice this will vary based on the resources available to the forger. Since most of the proof of work algorithms used by our protocols are *not* inherently sequential, the completion of the puzzles can be sped up by parallelizing the computation across more CPU cores. How to mitigate this problem, either through inherently sequential proofs of work or better guidelines for choosing a work factor δ, remains an open problem.

Chapter 4

Use Cases

In this chapter, we provide a few use cases for short-lived signatures (deniable email and instant messaging) and short-lived Σ-protocols (coercion-resistant, end-to-end verifiable voting).

Recall in Section 1 we briefly mentioned the principle of least privilege. There is little to be gained from allowing signatures to be attributed to a signer for longer than is strictly necessary. Traditional digital signatures already have a *de facto* expiration date that coincides with the expiration of the signing key. However these timelines for expiry are usually measured in years and are universal across all signatures made by a signing key. Most signatures only need to last a few seconds (in an interactive setting like a TLS handshake or instant messaging) or a few days (for non-interactive settings like email). By utilizing short-lived signatures with a suitable work factor as a drop in replacement for traditional digital signatures, we are able to facilitate deniability with minimal overhead.

4.1 Email

One of the simplest applications of a short-lived signature is email. Most scenarios in which a person would desire the security provided by a PGP signed or encrypted email are sensitive in nature and affording some deniability after the fact is likely to also be a desirable property.

By utilizing short-lived signed email, for example, a whistleblower would be able to reach out to a journalist such that the journalist can trust they're communicating with the same source, but after the fact the signatures on received messages are no longer trustworthy. A complementary approach, designated verifier signatures, would also allow for this type of deniability but require advanced knowledge of the receivers public key, which may not be known at the outset of an exchange. Once a public key has been exchanged, however, a short-lived DV signature could be used to further limit the verifiability of the signature to one or more parties as well as for a limited time.

4.2 Deniable Multiparty Messaging

4.2.1 Background

Off the record messaging (OTR) is an instant messaging protocol for end to end encrypted conversations between two parties. It aims to provide similar security properties to a private, in-person conversation, hence the name. In addition to confidentiality and authentication that other messaging protocols commonly provide, OTR also affords deniability.

In a face-to-face private conversation between two or more parties, once the conversation has ended neither party can prove that any particular statement was or was

not made or that the conversation even took place. In electronic protocols, signatures are often used to provide message integrity and to authenticate the sender. However, this leaves no room for deniability after the fact short of publishing the private keys, an approach we have already dismissed as unable to scale.

To permit deniability, OTR makes use of message authentication codes (MACs), a symmetric key primitive that verifies the integrity of a message. In a two party protocol like OTR, since MACs make use of symmetric keys they also authenticate the sending party of a message, much like a digital signature. If Alice sees a message with a valid MAC that she did not send, she knows it must have come from Bob since he is the only other person who knows the MAC key.

Unfortunately, since MACs can only provide authentication in this two party setting, there has been no obvious way to scale OTR to a group or multi-party setting. By using a short-lived signature instead of a MAC it could be possible to construct a multi-party OTR protocol in which the messages expire after an agreed upon amount of time.

There have been several attempts to define a multi-party version of OTR such as [BST07][GUVGC09]. [BST07] requires one user to act as a *virtual server*. This user, Alice, is trusted to act as a router for messages between all other parties and all of the other users, Bob and Carol, negotiate a MAC key with her. When sending a message to the group, Bob uses the MAC key he established with Alice and sends the message to her. Alice decrypts the message, and re-encrypts with MAC key for her and Carol before forwarding the message on to Carol as well. The amount of trust placed in the user chosen as the virtual server creates a massive power imbalance and the protocol can fall apart if this user acts dishonestly.

A more equitable approach is [GUVGC09]'s *mpOTR*, which makes use of ephemeral signing keys that are revealed at the end of every conversation. The authors also in-

troduced several useful sub-protocols for performing key exchange and agreement in a deniable fashion. The drawback to this approach is rooted in how the protocol defines the ending of a conversation. A user joining or leaving a conversation, at the protocol level, triggers a shutdown of the old conversation and creation of a new one with this user added or removed. In a high traffic chat room this may lead to keys being rotated far too frequently.

4.2.2 Using short-lived signatures

To adapt OTR short-lived signatures we could modify the mpOTR of protocol [GUVGC09] to make use of short-lived signatures instead of traditional signatures. This would remove the need for ephemeral signing keys in the protocol altogether since short-lived signatures do not require the private key to be released to allow forgeries to be created.

Recall that mpOTR also requires tearing down and setting up a new conversation whenever the set of participants in a group conversation changes. By using short-lived signatures instead, key rotation is no longer necessary and this overhead can be significantly reduced.

It is important to note that this produces a slightly different result than in other OTR protocols (two-party or the other multi-party proposals). In the other protocols, until the MAC (or ephemeral signing) key is leaked only the parties in the conversation can verify the authenticity of the message. Our short-lived approach also eliminates the need to explicitly leak the keys, though while the signature is valid there is no restriction on who is able to verify signatures. Practically this should not be of great concern since OTR is an interactive protocol. The parties communicating can use very short time periods to reduce the chances of a third party being able to verify a signature they were not intended to.

4.3 Voting

End-to-end verifiable (E2E) voting systems were introduced by Chaum in 1981 [Cha81]. These are voting systems that provide a universally verifiable proof that the tally was computed correctly and a voter verifiable proof that no ballots were modified or dropped. The challenge with these systems is protecting the ballot secrecy from the tallying authority, and further, preventing the voter from being able to prove to someone else how they voted.

4.3.1 Homomorphic tallying backbone

Consider the following template for an E2E scheme that is the backbone of dozens of schemes in the literature. A voter wishes to cast a vote for Alice in an election. To do this, she uses a voting machine that generates an encryption of Alice: $[\![A]\!]$. Assume the encryption scheme is additively homomorphic; that is, $[\![m_1]\!] \cdot [\![m_2]\!] = [\![m_1 + m_2]\!]$ for some efficient operation $\cdot$ such as multiplication. Exponential Elgamal [CGS97] and Paillier [Pai99] are examples of such a scheme. Assume the decryption key is shared amongst a set of n trustees (that are mutually distrustful) such that any $m \leq n$ of them can come together to decrypt a ciphertext encrypted under this shared key [Ped91]. For example, $m = 3$ and $n = 5$.

Assuming Alice can successfully get an encryption of the candidate she chooses (the critical issue we will return to) then the votes can be added up under encryption using the homomorphic property. In the simplest case of a two party race, votes for Alice might be encoded as 1 and votes for Bob as -1. The homomorphic property is used to sum up the encryptions and then the trustees convene to decrypt only this final summation of votes. A portion of the voting literature has been devoted to extending this idea to more than two candidates [HS00, Hir01, PAB+04, KY02]. Any

member of the public can validate that the individual encrypted votes sum to the same value that is decrypted, and the trustees can prove in zero knowledge that they are performing the decryption operation correctly without changing the value [Ped91].

The final question is how the voter can be confident that when she asks the voting machine for an encryption of Alice, it actually encrypts Alice and not Bob. Any proof she receives should be non-transferrable — that is, she should not be able to show it to Mallory as proof that she voted for Alice less Mallory be able to coerce her vote or to purchase it for money.

4.3.2 A solution based on DV proofs

The following system is a simplification of several ideas in the literature (*cf.* [MN06, DLM12]), put into terms of designated verifier proofs (see Section 2.3.3). Imagine the voter enters the voting booth with a keypair $\langle sk, pk \rangle$. Both keys might be printed as QR codes on a piece of paper. She scans the code for pk and asks for an encryption of Alice. The voting machine prints out $[\![A]\!]$ (where A is the encoding of Alice) which it asserts is correct. It then prints out a designated verifier zero knowledge proof, using pk, that has the following semantics: either $[\![A]\!]$ is an encryption of A; or I (the voting machine) know sk; or both. Since the voter has not revealed sk to the voting machine, she believes the first clause to be true. The voter then scans sk. The voting machine responds with a second proof of the following form: either $[\![B]\!]$ is an encryption of B; or I know sk; or both. The voter knows the first clause cannot be true, given the first proof, and therefore the second clause is true.

If the voter shows these proofs to Mallory, the proofs claim she voted for Alice and that she voted for Bob, and both are of the exact same form! Mallory cannot distinguish which is 'real.' The only distinguishing feature is the timing of the procedure. The 'real' proof is the one given to the voter before revealing sk. Since she

is in the voting booth at this time, only she knows which is 'real.'

4.3.3 Using short-lived proofs

The designated verifier scheme has one major drawback. Voters have to prepare a keypair to take into the booth. On paper, this may seem largely inconsequential. But in a real election with a wide range of voters — many of which are non-technical and may not own a computer or smartphone — we could expect a number of accessibility and usability issues. Voters may be unable to generate the keypair, may enter sk at the wrong time, enter sk instead of pk, *etc.*. It is also important that sk is truly secret or the voting machine could cheat.

We propose to simplify this scheme further with short-lived proofs. The basic voting procedure is largely the same as in the designated verifier scheme. The voter receives an encryption (asserted to be) of $[\![A]\!]$. This is printed on a sheet of paper that also includes the current time. The voter must confirm that the time is correct using her watch or smartphone. The voting machine immediately prints a short-lived zero knowledge proof (say that expires in 60 seconds) that $[\![A]\!]$ is an encryption of A. After 60 seconds, it prints a second proof (incorrect and simulated due to the expiration) that $[\![A]\!]$ is an encryption of B.

Once again, the voter has two contradicting proofs: one that shows a vote for Alice and one for Bob. The proofs are of exactly the same form and the only distinguishing feature is the timing of the proofs. The 'real' proof is the one that was issued within 60 seconds of the correct time and the 'fake' one(s) were issued afterward.

Chapter 5

Concluding Remarks

In this thesis, we introduced a new variety of digital signatures which only retain the non-repudiation property for a pre-determined amount of "time". We described four different short-lived schemes, one built on top of an existing digital signature scheme and three additional more concise constructions. We identified several desirable properties and evaluated our short-lived schemes against them. No *one* scheme managed to attain all of these properties, so it is likely that all of our proposed short-lived signatures will excel in different scenarios. As well as being a drop-in replacement for conventional digital signatures in most cases, we also described several potential use cases including two in instant messaging and voting that are uniquely suited to short-lived signatures and proofs.

Finally, our work blends the notion of real world time together with that of computational time. Much of the future work pertaining to short-lived signatures, such as preparing guidelines for how to choose an appropriate δ, is likely to rely on a better understanding of this interface.

Bibliography

[ANL00] T Aura, P Nikander, and J Leiwo. DoS-resistant authentication with client puzzles. In *Security Protocols*, 2000.

[Bac02] A Back. Hashcash: a denial of service counter-measure, 2002.

[BCG15] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On bitcoin as a public randomness source. *IACR Cryptology ePrint Archive*, 2015:1015, 2015.

[BdM91] J Benaloh and M de Mare. Efficient broadcast time-stamping. Technical Report TR-MCS-91-1, Clarkson University, 1991.

[BdM93] J Benaloh and M de Mare. One-way accumulators: a decentralized alternative to digital signatures. In *EUROCRYPT*, 1993.

[BHS91] D Bayer, S A Haber, and W S Stornetta. Improving the efficiency and reliability of digital time-stamping. In *Sequences*, 1991.

[BKZZ16] Foteini Baldimtsi, Aggelos Kiayias, Thomas Zacharias, and Bingsheng Zhang. Indistinguishable proofs of work or knowledge. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 902–933. Springer, 2016.

[BLLV98] A Buldas, P Laud, H Lipmaa, and J Villemson. Time-stamping with binary linking schemes. In *CRYPTO*, 1998.

[BN00] D Boneh and M Naor. Timed commitments. In *CRYPTO*, 2000.

[BPW12] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 626–643. Springer, 2012.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM, 1993.

[BST07] Jiang Bian, Remzi Seker, and Umit Topaloglu. Off-the-record instant messaging for group conversation. In *Information Reuse and Integration, 2007. IRI 2007. IEEE International Conference on*, pages 79–84. IEEE, 2007.

[CDNO97] Rein Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. Deniable encryption. In *Annual International Cryptology Conference*, pages 90–104. Springer, 1997.

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *CRYPTO*, 1994.

[CE12] Jeremy Clark and Aleksander Essex. Commitcoin: Carbon dating commitments with bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 390–398. Springer, 2012.

[CGS97] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *EUROCRYPT*, 1997.

[CH10] Jeremy Clark and Urs Hengartner. On the use of financial data as a random beacon. In *EVT/WOTE*, 2010.

[Cha81] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[CMSW09] L Chen, P Morrissey, N P Smart, and B Warinschi. Security notions and generic constructions for client puzzles. In *ASIACRYPT*, 2009.

[CP92] David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In *CRYPTO*, 1992.

[DLM12] Jérôme Dossogne, Frédéric Lafitte, and Olivier Markowitch. Coercion-freeness in e-voting via multi-party designated verifier schemes. In *EVOTE*, 2012.

[DMR06] S Doshi, F Monrose, and A D Rubin. Efficient memory bound puzzles using pattern databases. In *ACNS*, 2006.

[DN92] C Dwork and M Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1992.

[DS01] D Dean and A Subblefield. Using client puzzles to protect TLS. In *USENIX Security*, 2001.

[EP84] Shimon Even and Azaria Paz. A note on cake cutting. *Discrete Applied Mathematics*, 7(3):285–296, 1984.

[FM97] M K Franklin and D Malkhi. Auditable metering with lightweight security. In *Financial Cryptography*, 1997.

[GJMM98] E Gabber, M Jakobsson, Y Matias, and A Mayer. Curbing junk e-mail via secure classification. In *Financial Cryptography*, 1998.

[GS98] D M Goldschlag and S G Stubblebine. Publicly verifiable lotteries: Applications of delaying functions. In *Financial Cryptography*, 1998.

[GUVGC09] Ian Goldberg, Berkant Ustaoğlu, Matthew D Van Gundy, and Hao Chen. Multi-party off-the-record messaging. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 358–368. ACM, 2009.

[Hir01] Martin Hirt. *Multi-Party Computation: Efficient Protocols, General Adversaries and Voting*. PhD thesis, ETH Zurich, 2001.

[HS90] S Haber and W S Stornetta. How to time-stamp a digital document. In *CRYPTO*, 1990.

[HS00] Martin Hirt and Kazue Sako. Efficient receipt-free voting based on homomorphic encryption. In *EUROCRYPT*, 2000.

[JB99] A Juels and J Brainard. Client puzzles: A cryptographic defense against con- nection depletion attacks. In *NDSS*, 1999.

[JJ99] M Jakobsson and A Juels. Proofs of work and bread pudding protocols. In *Communications and Multimedia Security*, 1999.

[JSI96] Markus Jakobsson, Kazue Sako, and Russell Impagliazzo. Designated verifier proofs and their applications. In *EUROCRYPT*, 1996.

[KY02] Aggelos Kiayias and Moti Yung. Self-tallying elections and perfect ballot secrecy. In *PKC*, 2002.

[MB02] P Maniatis and M Baker. Enabling the long-term archival of signed documents through time stamping. In *FAST*, 2002.

[MMV11] M Mahmoody, T Moran, and S Vadhan. Time-lock puzzles in the random oracle model. In *CRYPTO*, 2011.

[MN06] Tal Moran and Moni Naor. Receipt-free universally-verifiable voting with everlasting privacy. In *CRYPTO*, 2006.

[Nak08] S Nakamoto. Bitcoin: A peer-to-peer electionic cash system. Unpublished, 2008.

[PAB+04] Kun Peng, Riza Aditya, Colin Boyd, Ed Dawson, and Byoungcheon Lee. Multiplicative homomorphic e-voting. In *INDOCRYPT*, 2004.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[Ped91] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In *EUROCRYPT*, 1991.

[PRQ+98] B Preneel, B Van Rompay, J J Quisquater, H Massias, and J S Avila. Design of a timestamping system. Technical Report WP3, TIMESEC Project, 1998.

[PS96] David Pointcheval and Jacques Stern. Security proofs for signature schemes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 387–398. Springer, 1996.

[Rab83] Michael Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences*, 27(2), 1983.

[RS96] R L Rivest and A Shamir. PayWord and MicroMint: two simple micropayment schemes. In *Security Protocols*, 1996.

[RSW96] R L Rivest, A Shamir, and D A Wagner. Time-lock puzzles and timed-release crypto. Technical Report TR-684, MIT, 1996.

[Sch91] C P Schnorr. Efficient signature generation by smart cards. *Journal of Cryptography*, 4, 1991.

[Sha92] Adi Shamir. Ip = pspace. *Journal of the ACM (JACM)*, 39(4):869–877, 1992.

[SKR+11] D Stebila, L Kuppusamy, J Rangasamy, C Boyd, and J M Gonzalez Nieto. Stronger difficulty notions for client puzzles and denial-of-service-resistant protocols. In *CT-RSA*, 2011.

[TBFG07] S Tritilanunt, C Boyd, E Foo, and J M Gonzalez Nieto. Toward non-parallelizable client puzzles. In *CANS*, 2007.

[WJHF04] B Waters, A Juels, J A Halderman, and E W Felten. New client puzzle outsourcing techniques for DoS resistance. In *CCS*, 2004.

[WR03] X Wang and M K Reiter. Defending against denial-of-service attacks with puzzle auctions. In *IEEE Symposium on Security and Privacy*, 2003.