An extended abstract version of this work appears in: Aggelos Kiayias, editor, *Topics in Cryptology – CT-RSA 2011 – The Cryptographers' Track at the RSA Conference, LNCS*, volume 6558, pp. 284–301. Springer, 2011. This is the full version.

Stronger difficulty notions for client puzzles and denial-of-service-resistant protocols
(full version)

Douglas Stebila Lakshmi Kuppusamy Jothi Rangasamy Colin Boyd
Juan Gonzalez Nieto
Information Security Institute, Queensland University of Technology, Brisbane, Australia
Email: {stebila, l.kuppusamy, j.rangasamy, c.boyd, j.gonzaleznieto}@qut.edu.au

December 21, 2010

Abstract

Client puzzles are meant to act as a defense against denial of service (DoS) attacks by requiring a client to solve some moderately hard problem before being granted access to a resource. However, recent client puzzle difficulty definitions (Stebila and Ustaoglu, 2009; Chen et al., 2009) do not ensure that solving n puzzles is n times harder than solving one puzzle. Motivated by examples of puzzles where this is the case, we present stronger definitions of difficulty for client puzzles that are meaningful in the context of adversaries with more computational power than required to solve a single puzzle.

A protocol using strong client puzzles may still not be secure against DoS attacks if the puzzles are not used in a secure manner. We describe a security model for analyzing the DoS resistance of any protocol in the context of client puzzles and give a generic technique for combining any protocol with a strong client puzzle to obtain a DoS-resistant protocol.

Keywords: client puzzles, proof of work, denial of service resistance, protocols

1 Introduction

Availability of services is an important security property in a network setting. *Denial of service (DoS) attacks* aim to disrupt the availability of servers and prevent legitimate transactions from taking place. One type of DoS attack is *resource depletion*: an attacker makes many requests trying to exhaust the server's resources, such as memory or computational power, leaving the server unavailable to service legitimate requests.

Client puzzles, also called *proofs of work*, can counter resource depletion DoS attacks. Before a server is willing to perform some expensive operation, it demands that the client commit some of its own resources by solving a puzzle. The puzzle should be moderately hard to solve – not as hard as a large factoring problem, for example, but perhaps requiring a few seconds of CPU time. Provided client puzzles are easy for a server to generate and verify, this creates an asymmetry between the amount of work done by a client and the work done by a server.

Although many client puzzle constructions have been proposed, there has been less work in rigourously defining good client puzzles or defining DoS resistance of protocols. The first model for client puzzles was proposed by Jakobsson and Juels [JJ99]. More recently, Stebila and Ustaoglu [SU09] described a security model for the DoS resistance of key exchange protocols, and Chen et al. [CMSW09a] proposed a formalization of client puzzles and puzzle difficulty, using a game between a single challenger and a single adversary.

1.1 Contributions and Outline

In this work, we motivate and present stronger notions of security for client puzzles and DoS resistance of protocols and provide several examples satisfying these new definitions.

An Attack on Previous Difficulty Definitions. The main motivation for our stronger notion of security is that it should be hard for an adversary to solve many puzzles, not just one. The existing DoS countermeasure models [JJ99, CMSW09a, SU09] address the ability of a runtime-bounded adversary to solve a single puzzle, but not of solving multiple puzzles: if one puzzle takes time 2^{20} to solve, for example, will 2^{30} puzzles will take time 2^{50} to solve? This is important in practice, for an adversary will likely have more power than needed to solve a single puzzle.

In order to demonstrate the inadequacy of existing definitions, in Sect. 2 we examine how for some puzzles – the generic puzzle construction of Chen et al. [CMSW09a], the MicroMint micropayment puzzle scheme [RS97], and number-theoretic puzzles such as the recent one of Karame-Čapkun [KC10] – it is hard to solve one instance (satisfying existing definitions [SU09, CMSW09a]), but many instances can be solved without too much more work. This is a weakness in the context of DoS resistance, and so a good puzzle difficulty definition should preclude this.

Stronger Client Puzzles. In Sect. 3, we propose two notions of strong difficulty for client puzzles, one for interactive situations and one for non-interactive situations. These stronger difficulty definitions capture the notion that solving n puzzles should cost about n times the cost of solving one puzzle. We then provide examples of puzzles satisfying these stronger definitions.

DoS-Resistant Protocols. In Sect. 4, we define what it means for a protocol to be DoS-resistant in a multi-user network setting. A server should not perform expensive operations unless a client has done the required work. It should be hard for the work of a legitimate client to be stolen or redirected (avoiding the attack of Mao and Paterson [MP02]). This generalizes the work of Stebila and Ustaoglu [SU09] on DoS-resistant key exchange protocols, while also accommodating our stronger notion of security for multiple puzzles as described above. Then, in Sect. 5, we present a theorem that shows how to transform any protocol into a DoS-resistant protocol using a strongly-difficult interactive client puzzle.

We conclude and discuss future work in Sect. 6. The appendices contain background definitions and notation (Appendix A) and proofs of claims from the main body (Appendices B–F).

1.2 Related Work

Client Puzzles. Client puzzles were first proposed for protection against DoS attacks (in the form of email spam) by Dwork and Naor [DN92]. Many client puzzle constructions have subsequently been proposed. There are two main types of client puzzles: *computation-bound* puzzles, which depend on a large number of CPU cycles to solve, and *memory-bound* puzzles [ABMW03, DGN03a, DNW05], which depend on a large number of memory accesses to solve, and which offer more uniform solving time across different CPU speeds compared to computation-bound puzzles. Many computation-bound puzzles are based on the difficulty of inverting a hash function [Bac97, JB99, JJ99, ANL00, Bac04, CMSW09a], although other techniques (for example, using number-theoretic primitives) exist as well [DN92, WJHF04, TBFGN06, KC10]. Puzzle-like constructions also appear in other cryptographic contexts [RSW96, RS97, Boy07] but with a focus on different security properties.

Difficulty of Client Puzzles. Although there have been many puzzle constructions as noted above, only a few of these use any formal notion of security, and there has been little work in developing formal definitions of client puzzle difficulty. The first client puzzle difficulty definition was given by Jakobsson and Juels [JJ99], and another by Canetti et al. [CHS05]. Some memory-bound puzzles [DGN03a, DNW05] include proofs of amortized difficulty.

A richer difficulty definition was given by Chen et al. [CMSW09a], using two security experiments: unforgeability and puzzle difficulty. Importantly, the difficulty definition only addresses the ability of an adversary to solve a single puzzle. They describe a basic generic client puzzle protocol $\Pi(\mathsf{CPuz})$. Finally, they give a generic client puzzle construction from a pseudorandom function and a one-way function (essentially a MAC and a hash function).

Our definition of puzzle difficulty starts from the Chen et al. [CMSW09a] definition, but with a number of differences. First, we eliminated the unforgeability property. The unforgeability property is important for their protocol $\Pi(\mathsf{CPuz})$, but is not an essential feature of client puzzles. In fact, to define non-interactive puzzles, in which the client can generate the puzzle itself, we must remove unforgeability. Next, we strengthened the difficulty definition to consider an adversary who solves many puzzles, motivated by our attack in Sect. 2. Our DoS resistance model and protocol is significantly stronger than their protocol $\Pi(\mathsf{CPuz})$, accommodating multiple users in a network setting.

Multiple Puzzles. Our work is motivated by the difficulty of solving multiple puzzles which has not been addressed adequately in previous works. Jakobsson and Juels [JJ99] considered independence of proofs of work, but only in terms of their solvability, not their difficulty. Canetti et al. [CHS05] addressed hardness amplification – the difficulty of solving many instances – of *weakly verifiable puzzles* (WVPs), which are puzzles that need not be publicly verifiable. The adversary for WVPs could not see valid puzzle/solution pairs, so Dodis et al. [DIJK09] introduced *dynamic* WVPs that did allow the adversary to see solutions and gave a hardness amplification theorem showing that if solving one dynamic WVP is hard, then solving an n-wise dynamic WVP is also hard. Still, dynamic WVPs differ from the difficulty definition of Chen et al. [CMSW09a] and our definition, in that dynamic WVPs generate all puzzle challenges at once, independent of the request, whereas puzzles in the Chen et al. model are generated in response to, and are dependent upon, client requests.

Modelling DoS Attacks on Protocols. Meadows [Mea99] presented a cost-based framework for identifying DoS attacks in network protocols (e.g., Smith et al.'s DoS attack [SGNB06] on the JFK key exchange protocol [ABB$^+$04]), but can only be used to identify and quantify a DoS attack, not prove that a protocol is DoS-resistant.

Stebila and Ustaoglu [SU09] gave a provable security model for the DoS resistance of key agreement protocols based on the eCK model for key agreement security [LLM07]. The model splits key exchange into two portions: a presession for the DoS countermeasure, and a session for the key exchange. They give an example protocol using hash function inversions for the DoS countermeasure and building on CMQV [Ust08] for the key exchange protocol. One of their main motivations was to avoid the DoS attack of Mao and Paterson [MP02] which derived from an authentication failure where messages could be redirected and accepted.

Our definition of DoS resistance for protocols shares some of these characteristics: it uses a presession for the DoS countermeasure and is suitable for a multi-user network setting. It can be used to analyze all protocols, not just key exchange protocols, and it uses a stronger notion of security, considering an adversary who solves many puzzles, not just one. By separating the definition of a puzzle from the definition of a DoS-resistant protocol, we can perform a modular analysis of each component separately and then combine them.

2 Weaknesses in Existing Definitions

In a public network setting, a server will be providing service to many clients at a time. A DoS countermeasure based on client puzzles should require appropriate work to be done for *each* client request: it should not be possible to solve many puzzles easily. While the existing models [JJ99, SU09, CMSW09a] describe the difficulty of DoS countermeasures when faced with an adversary trying to solve one puzzle, these models do not adequately defend against *powerful adversaries* who can expend more than the effort required to solve a single puzzle.

In this section, we consider some puzzles where a single instance cannot be solved easily by an attacker, satisfying existing difficulty definitions, but where an attacker can solve n puzzles more efficiently than just n times the cost of solving a single puzzle. This motivates our stronger definition of puzzle difficulty in Sect. 3.

While the examples in this section focus on the security definition of Chen et al. [CMSW09a], they can also be applied to the model of Stebila and Ustaoglu [SU09].

Generic Puzzle Construction of Chen et al. Chen et al. [CMSW09a] proposed a generic client puzzle construction based on a pseudorandom function $\mathcal{F}$ and a one-way function ϕ. The challenger selects a secret $s \in \mathcal{K}$ with $|\mathcal{K}| = 2^k$ and public parameters (not relevant to our discussion here), denoted by $*$, to generate a puzzle. The challenger computes $x \leftarrow \mathcal{F}(s, *)$, where $x \in \mathcal{X}$ and $|\mathcal{X}| \geq |\mathcal{K}|$, and then sets $y \leftarrow \phi(x)$. The solver, given the challenge $(y, *)$, has to find a pre-image z such that $\phi(z) = y$.

This generic construction satisfies the puzzle unforgeability and puzzle difficulty security properties provided certain bounds are met: namely, $|\mathcal{X}| \geq |\mathcal{K}|$ and $\frac{|\phi^{-1}(y)|}{|\mathcal{X}|} \leq \frac{1}{2^k}$, for all y. Suppose we have that $|\phi^{-1}(y)| \leq 1$ and $|\mathcal{X}| = 2^k$. Then the bounds in the generic construction are satisfied and solving a single puzzle instance requires approximately 2^k searches in $\mathcal{X}$. But to solve n puzzles, the solver can find the value s with at most 2^k searches and then obtain a solution with one application of $\mathcal{F}$ for each puzzle. That is, solving n puzzles would require $2^k + n$ operations rather than the desired $n \cdot 2^k$ computations.

MicroMint-Based Puzzle. The MicroMint micropayment scheme [RS97] is effectively a client-puzzle-based micropayment scheme. A coin is a collision in a hash function: it is a pair of values x_1, x_2 such that $H(x_1) = H(x_2)$ for a given hash function H. It is easy to verify the validity of a coin.

Generating coins is harder. If H is a regular (or random) function with ℓ-bit outputs, then to find a collision one must rely on the "birthday paradox" (c.f. [Sti02, §4.2.2]): hash approximately $2^{\ell/2}$ distinct values and search for a collision. This puzzle can be shown to satisfy the puzzle difficulty definition of the Chen et al. model [CMSW09a] (see Appendix Bfor details).

However, many collisions can be found without too much more work: n collisions can be found with $\sqrt{n} \cdot 2^{\ell/2}$ hash function calls, much less than n times the $2^{\ell/2}$ cost of solving a single puzzle. We emphasize this is not an attack on the MicroMint scheme itself: MicroMint was in fact *designed* so that the amortized cost of generating multiple coins is smaller. While potentially a desirable property in a micropayment scheme, this property is not desirable for client puzzles.

Number-Theoretic Puzzles. Many client puzzles based on number-theoretic constructions have been presented, including the recent scheme of Karame and Čapkun [KC10], which uses modular exponentiation and argues for security in the Chen et al. model [CMSW09a] based on the intractability of the RSA problem. Given a puzzle consisting of an RSA modulus N, a challenge x, and a large integer

$R >> N$, the solver must compute $x^R \mod N$.

The security argument rests on the assumption that the best known algorithm for this computation requires $O(\log(R))$ modular operations, assuming that factoring N requires more than $O(\log(R))$ operations. For a common puzzle difficulty level of say 2^{20}, a 1024-bit modulus N certainly suffices. But in fact a much smaller N would still suffice and would reduce the computational costs for the verifier, which is important when puzzles are used at extremely low levels in the network stack, such as TCP (e.g., as in [MPM04]).

Even with a smaller N, say 500 bits, the cost of solving a puzzle by computing $x^R \mod N$ is still cheaper than factoring (2^{20} compared to approximately 2^{49} based on the formulas in [BCC+08, §6.2]). However, if the adversary wants to solve 2^{30} puzzles, the best technique is *not* to solve all these puzzles independently (at a cost of $2^{30} \cdot 2^{20} = 2^{50}$ operations) but to first factor N and then use this trapdoor to easily generate solutions (at a cost of $2^{49} + 2^{30}c < 2^{50}$, for some small c which is the cost of easily generating solutions).

Signature forgery. In Appendix C, we present another counterexample puzzle based on signature forgery.

3 Strong Client Puzzles

The starting point for our definition of strong client puzzles is the model of Chen et al. [CMSW09a]. The main differences are as follows.

Firstly, as motivated by Sect. 2, our definition of puzzle difficulty is more robust in that it considers the number of puzzles solved by powerful adversaries.

Secondly, we omit the unforgeability security notion for client puzzles. Inherently, there is no need for puzzles to be unforgeable: in a game played between a challenger and an adversary, the challenger can keep track of all the puzzles issued to detect any forgeries. It is only when *using* puzzles in network protocol that unforgeability sometimes becomes relevant. The main purpose of unforgeability in Chen et al. [CMSW09a] was to show the DoS resistance of their client puzzle protocol construction $\Pi(\mathsf{Puz})$. We argue in Sect. 4 that a richer notion of DoS resistance is required for a multi-user network setting.

Thirdly, our puzzle definition ensures that the puzzle's semantic meaning – represented by the string str, which may identify the resource the client wishes to access – is the same for both the solver and the verifier. In the model of Chen et al. [CMSW09a], the server's generation of *puz* depended on str, but not in a way that the client could verify: *puz* was an opaque data structure. Thus, a client solving *puz* could not be certain that this would gain access to the str resource; and similarly, a server receiving a solution for *puz* could not know that the client solving *puz* intended to solve a puzzle related to str. This could allow client's work to be stolen by an attacker [SU09] or redirected [MP02]. By making a connection between str and *puz* more transparent, we can incorporate semantic meaning from other protocols or applications into a puzzle.

Fourthly, our security experiment allows for non-publicly verifiable puzzles, as suggested in the notion of weakly verifiable puzzles [CHS05].

Finally, in order to accommodate a variety of puzzle uses, we define two types of difficulty experiments, one for interactive settings and one for non-interactive settings. This accommodates asynchronous applications, such as email, where the client itself generates the puzzle [Bac97, Bac04]. While the non-interactive definition is more general, it is often convenient to consider the more limited interactive definition because of its simplicity and its more natural use in interactive protocols. We provide examples of puzzles satisfying each type, and interactive puzzles are at the heart of our DoS-resistant protocol construction in Sect. 5.

3.1 Client Puzzles

Definition 1 (Client Puzzle)**.** *A* client puzzle Puz *is a tuple consisting of the following algorithms:*

- $\mathsf{Setup}(1^k)$ *(p.p.t. setup algorithm):*
    1. *Choose the long-term secret key space* sSpace*, puzzle difficulty space* $\mathsf{diffSpace}$*, string space* $\mathsf{strSpace}$*, puzzle space* $\mathsf{puzSpace}$*, and solution space* $\mathsf{solnSpace}$.
    2. *Set* $s \leftarrow_R \mathsf{sSpace}$.
    3. *Set* $params \leftarrow (\mathsf{sSpace}, \mathsf{puzSpace}, \mathsf{solnSpace}, \mathsf{diffSpace}, \Pi)$*, where* Π *is any additional public information, such as a description of puzzle algorithms, required for the client puzzle.*
    4. *Return* $(params, s)$.
- $\mathsf{GenPuz}(s \in \mathsf{sSpace}, d \in \mathsf{diffSpace}, str \in \mathsf{strSpace})$ *(p.p.t. puzzle generation algorithm): Return* $puz \in \mathsf{puzSpace}$.
- $\mathsf{FindSoln}(str \in \mathsf{strSpace}, puz \in \mathsf{puzSpace}, t \in \mathbb{N})$ *(probabilistic solution finding algorithm): Return a potential solution* $soln \in \mathsf{solnSpace}$ *after running time at most* t.[1]
- $\mathsf{VerSoln}(s \in \mathsf{sSpace}, str \in \mathsf{strSpace}, puz \in \mathsf{puzSpace}, soln \in \mathsf{solnSpace})$ *(d.p.t. puzzle solution verification algorithm): Returns* **true** *or* **false**.

For *correctness*, we require that if $(params, s) \leftarrow \mathsf{Setup}(1^k)$ and $puz \leftarrow \mathsf{GenPuz}(s, d, str)$, for $d \in \mathsf{diffSpace}$ and $str \in \mathsf{strSpace}$, then there exists $t \in \mathbb{N}$ with

$$\Pr\left(\mathsf{VerSoln}(s, str, puz, soln) = \mathbf{true} : soln \leftarrow \mathsf{FindSoln}(str, puz, t)\right) = 1 \ .$$

3.2 Strong Puzzle Difficulty

A puzzle satisfies strong puzzle difficulty if the probability that a runtime-bounded-adversary can output a list of n fresh, valid puzzle solutions is upper-bounded by a function of the puzzle difficulty parameter and n. This is formalized in the following two experiments for the interactive and non-interactive settings.

We first need to define additional helper oracles as follows:

- $\mathsf{GetPuz}(str)$: Set $puz \leftarrow \mathsf{GenPuz}(s, d, str)$ and record (str, puz) in a list. Return puz.
- $\mathsf{GetSoln}(str, puz)$: If (str, puz) was not recorded by GetPuz, then return $\perp$. Otherwise, find $soln$ such that $\mathsf{VerSoln}(s, str, puz, soln) = \mathbf{true}$. Record $(str, puz, soln)$. Return $soln$.[2]
- $\mathsf{V}(str, puz, soln)$: Return $\mathsf{VerSoln}(s, str, puz, soln)$.

3.2.1 Interactive Strong Puzzle Difficulty

In this setting, we imagine a solver interacting with a challenger: the solver submits a request for a puzzle, the challenger issues a puzzle, the solver sends a solution to the challenger, and the challenger checks the solution. The solver can only submit solutions to puzzles that were issued by the challenger:

[1] $\mathsf{FindSoln}$ runs in time *at most* t so that a client can stop searching for a puzzle after a specified amount of time; the difficulty definitions in Sect. 3.2 yield that a client must spend *at least* a certain amount of time to find a valid solution.

[2] Note that $\mathsf{GetSoln}$ is only obligated to find a solution if puz was actually generated by the challenger. If $\mathcal{A}$ generated puz, then $\mathcal{A}$ may need to employ $\mathsf{FindSoln}$ to find a solution. Compared to $\mathsf{FindSoln}$, $\mathsf{GetSoln}$ has access to additional secret information that may allow it to find a solution more easily.

this immediately rules out puzzle forgery or generation of puzzles by the solver. The challenger also allows the solver, via queries, to see solutions to other puzzles.

Let k be a security parameter, let d be a difficulty parameter, let $n \geq 1$, and let $\mathcal{A}$ be an algorithm. The security experiment $\mathsf{Exec}^{\text{INT-STR-DIFF}}_{\mathcal{A},d,\mathsf{Puz}}(k)$ for interactive strong puzzle difficulty of a puzzle Puz is defined as follows:

- $\mathsf{Exec}^{\text{INT-STR-DIFF}}_{\mathcal{A},n,d,\mathsf{Puz}}(k)$:
  1. Set $(params, s) \leftarrow \mathsf{Setup}(1^k)$.
  2. Set $\{(str_i, puz_i, soln_i) : i = 1, \ldots, n\} \leftarrow \mathcal{A}^{\mathsf{GetPuz},\mathsf{GetSoln},\mathsf{V}}(params)$.
  3. If $\mathsf{VerSoln}(s, str_i, puz_i, soln_i) = \textbf{true}$, the tuple (str_i, puz_i) was recorded by GetPuz, and $(str_i, puz_i, soln_i)$ was not recorded by $\mathsf{GetSoln}$ for all $i = 1, \ldots, n$, then return **true**, otherwise return **false**.

Definition 2 (Interactive Strong Puzzle Difficulty)**.** *Let $\epsilon_{d,k,n}(t)$ be a family of functions monotonically increasing in t, where $\epsilon_{d,k,n}(t) \leq \epsilon_{d,k,1}(t/n)$ for all t, n such that $\epsilon_{d,k,n}(t) \leq 1$. Fix a security parameter k and difficulty parameter d. Let $n \geq 1$. Then* Puz *is an* $\epsilon_{d,k,n}(\cdot)$-strongly-difficult interactive client puzzle *if, for all probabilistic algorithms $\mathcal{A}$ running in time at most t,*

$$\Pr\left(\mathsf{Exec}^{\text{INT-STR-DIFF}}_{\mathcal{A},n,d,\mathsf{Puz}}(k) = \textbf{true}\right) \leq \epsilon_{d,k,n}(t) \enspace .$$

In the random oracle model,[3] To our knowledge, this is the first formal justification for the security of Hashcash. we can define interactive and non-interactive strong puzzle difficulty in terms of the number of oracle queries made by the adversary instead of its running time.

Remark. The condition that $\epsilon_{d,k,n}(t) \leq \epsilon_{d,k,1}(t/n)$, for all t and n such that $\epsilon_{d,k,n}(t) \leq 1$, captures the property that solving n puzzles should cost n times the cost of solving one puzzle, at least until the adversary spends enough time t to solve n puzzles with probability 1.

Remark. This bound is quite abstract; let us consider a concrete function for $\epsilon_{d,k,n}(t)$. For example, suppose each Puz instance should take approximately 2^d steps to solve. Then we might aim for Puz to be a $\epsilon_{d,k,n}(\cdot)$-strongly-difficult interactive client puzzle, where $\epsilon_{d,k,n}(t) \approx t/2^d n + \mathrm{negl}(k)$.

Remark. In the security experiment, the adversary is allowed to request many more than n puzzles using GetPuz. The adversary can then pick which n puzzles it submits as its allegedly solved puzzles $\{(str_i, puz_i, soln_i) : i = 1, \ldots, n\}$. In other words, the adversary could request many puzzles and hope to find some easy-to-solve instances. This means, for example, that puzzles for which 1% of instances are trivially solved could not be proven secure (with a reasonable $\epsilon_{d,k,n}(t)$) according to this difficulty definition.

Relation to Examples from Sect. 2. The Chen et al. generic puzzle construction in Sect. 2 does not satisfy our definition of strong puzzle difficulty. From Theorem 2 of [CMSW09a], we have that the Chen et al. generic construction is $\epsilon_{d,k}(t)$-difficult, with $\epsilon_{d,k}(t) \lessapprox 2\nu_k(t) + (1 + t/(2^{k-t}))\gamma_d(t)$, where $\nu_k(t)$ is the probability of breaking the pseudorandom function family (with security parameter k) in time t and $\gamma_d(t)$ is the probability of breaking the one-way function (with security parameter d) in time t. By the argument from Sect. 2, there exists an adversary that can win the strongly-difficulty interactive puzzle game with probability at least $\epsilon'_{d,k,n}(t) \gtrapprox \nu_k(t) + \gamma_d(t)/n$, which does not satisfy $\epsilon'_{d,k,n}(t) \leq \epsilon'_{d,k,1}(t/n)$.

[3] In the *random oracle model*, a hash function is modelled as an ideal random function accessible to the adversary solely as an oracle. [BR93c]

Similarly, the MicroMint puzzle from Sect. 2 does not satisfy Definition 2. Finding a single ℓ-bit collision (and thus solving a MicroMint puzzle) requires about $2^{\ell/2}$ hash function calls, but finding n collisions requires only $\sqrt{n} \cdot 2^{\ell/2}$ calls. Let $\epsilon_{k,\ell,n}(q) = \frac{q}{\sqrt{n} \cdot 2^{\ell/2}}$. It is clear that, for $n \geq 2$, $\epsilon_{k,\ell,n}(q) > \epsilon_{k,\ell,1}(q/n)$, and hence MicroMint is not an $\epsilon_{k,\ell,n}(\cdot)$-strongly difficulty interactive puzzle.

Similarly, the Karame-Čapkun puzzle [KC10] does not satisfy the interactive strong puzzle difficulty definition since for sufficiently many puzzles the best approach is to factor the RSA modulus N and use the trapdoor information to quickly solve puzzles. In other words, $\epsilon_{d,k,n}(t)$ is not less than or equal to $\epsilon_{d,k,1}(t/n)$ for sufficiently large n.

3.2.2 Non-Interactive Strong Puzzle Difficulty

Non-interactive strong puzzle difficulty models the case of client-generated puzzles. Besides being useful in their originally proposed setting as an email spam countermeasure [Bac97, Bac04], they can be useful in protocols that are inherently asynchronous, such as the Internet Protocol (IP), or have a fixed message flow, such as the Transport Layer Security (TLS) protocol.

The technical difference between interactive and non-interactive strongly difficult puzzles is whether the adversary can return solutions only to puzzles generated by the challenger (interactive) or can also return solutions to puzzles it generated itself (non-interactive).

The security experiment $\mathsf{Exec}^{\text{NINT-STR-DIFF}}_{\mathcal{A},n,d,\mathsf{Puz}}(k)$ for non-interactive strong puzzle difficulty is as in the interactive case with a change to line 3 of the experiment:

- $\mathsf{Exec}^{\text{NINT-STR-DIFF}}_{\mathcal{A},n,d,\mathsf{Puz}}(k)$:
    3. If $\mathsf{VerSoln}(s, str_i, puz_i, soln_i) =$ **true** and the tuple $(str_i, puz_i, soln_i)$ was not recorded by $\mathsf{GetSoln}$ for all $i = 1, \ldots, n$, then return **true**, otherwise return **false**.

The definition of $\epsilon_{d,k,n}(\cdot)$*-strongly-difficult non-interactive client puzzles* follows analogously.

Remark. If Puz is an $\epsilon_{d,k,n}(\cdot)$-strongly-difficult non-interactive puzzle, then it is also $\epsilon_{d,k,n}(\cdot)$-strongly-difficult interactive puzzle.

3.3 A Strongly-Difficult Interactive Client Puzzle Based on Hash Functions

In this section, we describe a client puzzle based on hash function inversion, similar to the subpuzzle used by Juels and Brainard [JB99] or the partial inversion proof of work of Jakobsson and Juels [JJ99].

Let $H : \{0,1\}^* \to \{0,1\}^k$ be a hash function. Define $\mathbf{SPuz}_H$ be the following tuple of algorithms:

- $\mathsf{Setup}(1^k)$: Set $\mathsf{sSpace} \leftarrow \{\perp\}$, $\mathsf{diffSpace} \leftarrow \{0,1,\ldots,k\}$, $\mathsf{strSpace} \leftarrow \{0,1\}^*$, $\mathsf{puzSpace} \leftarrow \{0,1\}^* \times \{0,1\}^k$, $\mathsf{solnSpace} \leftarrow \{0,1\}^*$, and $s \leftarrow \perp$.
- $\mathsf{GenPuz}(\perp, d, str)$: Set $x \leftarrow_R \{0,1\}^k$; let x' be the first d bits of x and x'' be the remaining $k-d$ bits of x. Set $y \leftarrow H(x, d, str)$. Return $puz \leftarrow (x'', y)$.
- $\mathsf{FindSoln}(str, (x'', y), t)$: For z from 0 to $\max\{t, 2^d - 1\}$: set $soln \leftarrow z$ (in $\{0,1\}^d$); if $H(soln \| x'', d, str) = y$ then return $soln$.
- $\mathsf{VerSoln}(\perp, str, (x'', y), soln)$: If $H(soln \| x'', d, str) = y$ then return **true**, otherwise return **false**.

Theorem 1. *Let H be a random oracle. Let $\epsilon_{d,k,n}(q) = \left(\frac{q+n}{n2^d}\right)^n$. Then $\mathbf{SPuz}_H$ is an $\epsilon_{d,k,n}(q)$-strongly-difficult interactive client puzzle, where q is the number of distinct queries to H.*

The proof follows a counting argument and appears in Appendix D.[4]

[4] Rather than proceeding directly to the random oracle model, we could aim to prove $\mathbf{SPuz}_H$ secure when H has some concrete hash function property. None of the standard hash function notions [RS04a] is appropriate due to (a) the partial preimage hint x'' being given and (b) the multiple nature of the task. One could extend the partial-inversion proof of work

3.4 Hashcash is a Strongly-Difficult Non-interactive Client Puzzle

In this section, we show that one of the earliest client puzzles, Hashcash [Bac97, Bac04], satisfies the definition of a strongly-difficult non-interactive client puzzle in the random oracle model.

While Hashcash was originally proposed to reduce email spam, the current specification (stamp format version 1 [Bac04]) can be applied to any resource. Hashcash is non-interactive: the puzzle is generated by the same person who solves the puzzle. Hence it should be difficult for a client to generate a puzzle that can be easily solved. Hashcash is based on the difficulty of finding a partial preimage of a string starting with a certain number of zeros in the SHA-1 hash function.

A Hashcash *stamp* is a string of the form `ver:bits:date:resource:[ext]:rand:counter`. The field `bits` denotes the "value" of the stamp (the number of zeros at the start of the output) and `counter` is the solution to the puzzle. A stamp is *valid* if $H(stamp)_{[1\ldots\mathtt{bits}]} = 0\ldots0$. In the context of real-world email applications, there may be additional restrictions on the validity of a stamp, such as whether `date` is within a reasonable range and whether the email address (`resource`) specified is acceptable.

Let $\mathbf{Hashcash}_H$ be the specification of the Hashcash puzzle using the hash function $H : \{0,1\}^* \to \{0,1\}^k$ in the language of Sect. 3.1. The precise specification is omitted here and given in Appendix E, but it proceeds in the obvious way. In particular, we note that Hashcash requires no long-term secret key (so $\mathsf{sSpace} = \{\perp\}$).

Theorem 2. *Let $H : \{0,1\}^* \to \{0,1\}^k$, where $k \geq d$, be a random oracle. Let $\epsilon_{d,k,n}(q) = \frac{q+n}{n2^d}$. Then Hashcash is an $\epsilon_{d,k,n}(q)$-strongly-difficult non-interactive puzzle, where q is the number of queries made by $\mathcal{A}$ to H.*

The proof follows a counting argument and appears in Appendix E.

4 Denial-of-Service Resistance of Protocols

Although we have defined what a good client puzzle is, it does not immediately follow that using a good client puzzle in a protocol yields DoS resistance. In this section, we describe what it means for a protocol to be DoS-resistant, and in the subsequent section we give a generic construction for DoS-resistant protocols.

Our approach begins similar to that of Stebila and Ustaoglu [SU09]. We work in an adversary-controlled multi-user communication network.[5] The adversary's goal is to cause a server to commit resources without the adversary itself having done the work to satisfy the denial of service countermeasure.

Protocol Execution. A protocol is a message-driven interaction, taking place among disjoint sets of clients $\mathsf{Clients}$ and servers $\mathsf{Servers}$, where each *party* is a probabilistic polynomial-time Turing machine. An execution of the protocol is called a *presession*. During execution, each party $\hat{U}$ may have multiple instances of the protocol running, with each instance indexed by a value $i \in \mathbb{Z}_+$; these instances are denoted by $\Pi_i^{\hat{U}}$. A protocol consists of the following algorithms:

notion [JJ99] (which satisfies (a) but not (b)) or the ePre notion [RS04a] (which satisfies neither) as appropriate, and then proceed to the random oracle model to heuristically justify the soundness of that new notion; the end result would be the same.

[5] It is true that, in an adversary-controlled network, the adversary can deny service simply by not relaying messages. Our concern, however, is with resource depletion attacks in which a server is overwhelmed with requests.

- $\mathsf{GlobalSetup}(1^k)$ (p.p.t. protocol setup algorithm): Select the long-term secret key space $\rho\mathsf{Space}$. Choose global public parameters Π of the scheme and return $params \leftarrow (\rho\mathsf{Space}, \Pi)$; this is assumed to be an implicit input to all remaining algorithms.
- $\mathsf{ServerSetup}(\hat{S} \in \mathsf{Servers})$ (p.p.t. party setup algorithm): Select $\rho_{\hat{S}} \in \rho\mathsf{Space}$. Perform any additional setup required by $params$.
- $\mathsf{CAction}j(\hat{C} \in \mathsf{Clients}, i \in \mathbb{Z}_+, m_{j-1}, M'_{j-1})$, for $j = 1, \ldots$ (p.p.t. protocol client action algorithm): Instance i of party $\hat{C}$ produces its jth protocol message for the run of the protocol, based on the instance's previous private state m_{j-1} and the received message M'_{j-1}. The output (M_j, m_j) consists of its outgoing message M_j and its new private state m_j.
- $\mathsf{SAction}j(\hat{S} \in \mathsf{Servers}, i \in \mathbb{Z}_+, m'_{j-1}, M_j)$, for $j = 1, \ldots$ (p.p.t. protocol server action algorithm): Instance i of party $\hat{S}$ produces its jth protocol message for this instance, based on $\hat{S}$'s long-term secret, the previous private state m'_{j-1}, and the received message M_j. The output (M'_j, m'_j) consists of its outgoing message M'_j and its new private state m'_j.

The client is assumed to be the initiator. An instance records its current progress through the protocol with the value j of the last completed action.

Presessions. After receiving some sequence of $\mathsf{SAction}j(\hat{S}, i, \ldots)$ calls, a server instance will either *accept* or *reject*; if it accepts, it outputs a *presession* identified by a tuple of the form $[\hat{C}, \hat{S}, \tau]$, where $\hat{C}$ is the *partner* and τ is a sequence of messages. The sequence of messages τ is meant to act like a transcript; however, since in DoS-resistant protocols a server may not store state early in the protocol, portions of τ could have been forged by an adversary. Accepted presessions must be unique within a party. Additionally, since the protocol may be used for another purpose – key agreement, electronic voting, etc. – we do not require that the protocol terminate after accepting, and indeed expect that it may continue to perform some additional application-level functionality.

Correctness. A protocol is *correct* if, for all $\hat{C} \in \mathsf{Clients}$ and $\hat{S} \in \mathsf{Servers}$ who follow the protocol, there exists a running time t for $\hat{C}$ such that, when messages are relayed faithfully between $\hat{C}$ and $\hat{S}$, $\hat{S}$ will accept with probability 1. In other words, clients can eventually do enough work to make connections.[6]

Denial of Service Countermeasure. To provide DoS resistance, a protocol will typically include some test so the server can decide, based on the proposed presession $[\hat{C}, \hat{S}, \tau]$ and its secret ρ, whether to accept or reject based on some DoS countermeasure in the protocol. It is the adversary's goal to cause a server to accept without the adversary having faithfully followed the protocol.

Adversary's Powers. The adversary controls all communication links and can send, create, modify, delay, and erase messages to any participants. Additionally, the adversary can learn private information from parties or cause them to perform certain actions.

The following queries model how the adversary interacts with the parties:

- $\mathsf{Send}(\hat{U}, i, M)$: The adversary sends message M to instance i of $\hat{U}$ who performs the appropriate protocol action (either $\mathsf{CAction}j(\hat{U}, i, m, M)$ or $\mathsf{SAction}j(\hat{U}, i, m, M)$ based on the instance's last completed action $j - 1$), updates its state, and returns its outgoing message, if any.
- $\mathsf{Expose}(\hat{S})$: The adversary obtains $\hat{S}$'s secret value $\rho_{\hat{S}}$; mark $\hat{S}$ as *exposed*.

Security Definition. The basic idea of the security definition is as follows: the amount of credit the adversary gets in terms of accepted presessions should not be greater than the amount of work the adversary itself did. An important part of the definition below is solutions from legitimate clients.

[6] Limits on the amount of work done by the server come later, in Definition 3.

An instance $\Pi_i^{\hat{S}}$ that has accepted a presession $[\hat{C}, \hat{S}, \tau]$ is said to be *fresh* provided that $\hat{S}$ was not exposed before $\hat{S}$ accepted this presession and there does not exist an instance $\Pi_j^{\hat{C}}$ which has a matching conversation [BR93a] for τ. (Intuitively, a "fresh" instance is an attackable instance, one that has not been trivially solved by exposing the server's private information.)

Let k be a security parameter, let $n \geq 1$, and let $\mathcal{A}$ be a probabilistic algorithm. The security experiment $\mathsf{Exec}^{\text{DOS}}_{\mathcal{A},n,P}(k)$ for DoS resistance of a protocol P is defined as follows:

- $\mathsf{Exec}^{\text{DOS}}_{\mathcal{A},n,P}(k)$: Run $\mathsf{GlobalSetup}(k)$. For each $\hat{S} \in \mathsf{Servers}$, run $\mathsf{ServerSetup}(\hat{S})$. Run $\mathcal{A}(params)$ with oracle access to Send and Expose. If, summing over all servers, the number of fresh instances accepted is n, then return **true**, otherwise return **false**.

A protocol is DoS-resistant if the probability that an adversary with bounded runtime can cause a server to accept n fresh presessions is bounded:

Definition 3 (Denial-of-service-resistant Protocol)**.** *Let $\epsilon_{k,n}(t)$ be a family of functions that are monotonically increasing in t, where $\epsilon_{k,n}(t) \leq \epsilon_{k,1}(t/n)$ for all t, n such that $\epsilon_{k,n}(t) \leq 1$. Fix a security parameter k. Let $n \geq 1$. We say that a protocol P is $\epsilon_{k,n}(\cdot)$-*denial-of-service-resistant *if*

1. *for all probabilistic algorithms $\mathcal{A}$ running in time at most t,*

$$\Pr\left(\mathsf{Exec}^{\text{DOS}}_{\mathcal{A},n,P}(k) = \textbf{true}\right) \leq \epsilon_{k,n}(t) + \mathrm{negl}(k) \text{ , and}$$

2. *no call to $\mathsf{SAction}j_P(\hat{S}, i, m, M)$ results in an expensive operation unless $\Pi_i^{\hat{S}}$ has accepted.*

Remark. This definition of DoS resistance contains two aspects. The first aspect addresses the ability of an adversary to cause the server to accept a presession: the inequality in part 1 provides a bound on the ability of an adversary to cause the server to accept n presessions when the adversary has only done t operations. The requirement that $\epsilon_{k,n}(t) \leq \epsilon_{k,1}(t/n)$ enforces the idea that the amount of work required to cause n presessions to be accepted should be n times the amount of work required to cause one presession to be accepted.

The second aspect addresses the idea that a server should not perform expensive operations unless the countermeasure has been passed. As the notion of "expensive" can vary from setting to setting, we leave it vague, but it can easily be formalized, for example by using Meadows' cost-based framework [Mea99].

Avoiding Client Impersonations. Though a DoS countermeasure does not provide explicit authentication, we still wish to avoid impersonations. For example, suppose a client $\hat{C}$ sends messages meant to prove its legitimate intentions in communicating with server $\hat{S}$. It should not be possible for an adversary to easily use those messages to cause another server $\hat{S}'$ to perform expensive operations, nor should it be possible for an adversary to easily use those messages to convince $\hat{S}$ that a different client $\hat{C}'$ intended to communicate with $\hat{S}$.

This is prevented by the model since party names are included in the presession identifiers. If an adversary observed a presession $[\hat{C}, \hat{S}, \tau]$ and then tried to use that information to construct a presession $[\hat{C}', \hat{S}, \tau']$ of another user $\hat{C}'$ with the same server, then this new presession would be unexposed and the adversary would be prohibited from easily causing a server to accept it by Definition 3. This in effect requires a binding of values in the DoS countermeasure transcript τ to the parties – $\hat{C}$ and $\hat{S}$ – in question.

Avoiding Replay Attacks. We follow the approach of Stebila and Ustaoglu [SU09] in dealing with replay attacks, where replay attacks are avoided by uniqueness of presession identifiers of accepted presessions. This does mean that the server has to store a table of presession identifiers, but this does not constitute a vector for a DoS attack because the server only stores a presession identifier after it accepts a presession, so it is doing an expensive operation only after the DoS countermeasure has been passed.

5 Building DoS-resistant Protocols from Client Puzzles

In this section, we present a generic technique that transforms any protocol P into a DoS-resistant protocol $D(P)$. Our technique uses strongly-difficult interactive client puzzles as a DoS countermeasure and message authentication codes for integrity of stateless connections [AN97]. We prove that the combined protocol $D(P)$ is a DoS-resistant protocol.

The client and server each provide nonces and construct the string str using their names, nonces, and any additional information, such as a timestamp or information from a higher-level protocol. The server generates a puzzle from str, authenticates the puzzle using the message authentication code (to avoid storing state), and sends it to the client. The client solves the puzzle using its own string str and sends the solution to the server. The server checks the message authentication code and the correctness of the solution. Finally, the server checks that the presession is unique and accepts. The messages for the DoS countermeasure are interleaved, where possible, with the messages of the main protocol, and after the countermeasure has accepted the main protocol continues as needed.

Specification. Let P be a protocol such that $\mathsf{SAction1}_P$ does not involve any expensive operations. Let k be a security parameter. Let $\mathsf{MAC} : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k$ be a family of secure message authentication codes (see Appendix A.2 for the full definition). Let $\mathsf{Puz} = (\mathsf{Setup}, \mathsf{GenPuz}, \mathsf{FindSoln}, \mathsf{VerSoln})$ be a strongly-difficult interactive client puzzle with long-term secret key space $\mathsf{sSpace} = \{\perp\}$ (there is no long-term secret key for puzzles). Although this may seem restrictive, many puzzles satisfy this constraint, including the hash-based puzzle in Sect. 3.3. Fix a DoS difficulty parameter $d \in \mathsf{diffSpace}$.

Let $D(P)_{\mathsf{Puz},d,\mathsf{MAC},k}$ be the protocol consisting of the following algorithms:

- $\mathsf{GlobalSetup}(1^k)$: Set $\rho\mathsf{Space} \leftarrow \{0,1\}^k$ and $\mathsf{NonceSpace} \leftarrow \{0,1\}^k$.
- $\mathsf{ServerSetup}(\hat{S} \in \mathsf{Servers})$: Set $mk_{\hat{S}} \leftarrow_R \{0,1\}^k$ and $\rho_{\hat{S}} \leftarrow mk_{\hat{S}}$.
- $\mathsf{CAction}j_{D(P)}(\dots), \mathsf{SAction}j_{D(P)}(\dots)$: As specified by the protocol in Figure 1.

Remark. The construction $D(P)$ requires that $\mathsf{SAction1}_P$ not involve any expensive operations, as $\mathsf{SAction1}_P$ is called by $\mathsf{SAction1}_{D(P)}$ before the server instance has accepted. If $\mathsf{SAction1}_P$ does in fact involve expensive operations, then P would need to be rewritten so that the expensive operation is delayed until $\mathsf{SAction2}_P$. In other words, the $D(P)$ construction may result in an additional round being added before the P protocol is run; this should not be surprising.

Additionally, $\mathsf{SAction1}_P$ may result in a private output m_1' which the server instance needs to store until the next message is received. If state storage is considered an expensive operation (as it could be a vector for a resource depletion DoS attack), then there are two options: use a stateless connection [AN97] to encrypt m_1' and send it to the client who must return it in the following round, or, as above, rewrite P so as to delay the operation until $\mathsf{SAction2}_P$.

Theorem 3. *Let P be a protocol such that $\mathsf{SAction1}_P$ does not involve any expensive operations. Suppose that Puz is an $\epsilon_{d,k,n}(t)$-strongly-difficult interactive puzzle with long-term secret key space $\mathsf{sSpace} = \{\perp\}$ and that MAC is a family of secure message authentication codes. Then $D(P)_{\mathsf{Puz},d,\mathsf{MAC},k}$ is an $\epsilon'_{d,k,n}(t)$-denial-of-service-resistant protocol, for $\epsilon'_{k,n}(t) = \epsilon_{d,k,n}(t + t_0 q_{\mathsf{Send}}) + \mathrm{negl}(k)$, where q_{Send} is the number*

$D(P)_{\mathsf{Puz},d,\mathsf{MAC},k}$ – $\mathsf{Send}(\hat{U},i,M)$ protocol specification

	Client $\hat{C}$		Server $\hat{S}$ long-term secret: $\rho_{\hat{S}} = mk_{\hat{S}}$
	$\underline{\mathsf{CAction1}_{D(P)}}$:		
1.	$N_C \leftarrow_R \mathsf{NonceSpace}$		
2.	$(M_1, m_1) \leftarrow \mathsf{CAction1}_P()$	$\xrightarrow{\hat{C},N_C,M_1}$	$\underline{\mathsf{SAction1}_{D(P)}}$:
3.			$N_S \leftarrow_R \mathsf{NonceSpace}$
4.			$(M_1', m_1') \leftarrow \mathsf{SAction1}_P(M_1)$
5.			$str \leftarrow (\hat{C}, \hat{S}, N_C, N_S, M_1, M_1')$
6.			$puz \leftarrow \mathsf{GenPuz}(\perp, d, str)$
7.	$\underline{\mathsf{CAction2}_{D(P)}}$:	$\xleftarrow{N_S,M_1',puz,\sigma}$	$\sigma \leftarrow \mathsf{MAC}_{mk_{\hat{S}}}(str, puz)$
8.	$str \leftarrow (\hat{C}, \hat{S}, N_C, N_S, M_1, M_1')$		
9.	$soln \leftarrow \mathsf{FindSoln}(str, puz, t)$		
10.	$(M_2, m_2) \leftarrow \mathsf{CAction2}_P(m_1, M_1')$	$\xrightarrow{str,puz,\sigma,soln}$	$\underline{\mathsf{SAction2}_{D(P)}}$:
11.			reject if $\sigma \neq \mathsf{MAC}_{mk_{\hat{S}}}(str, puz)$
12.			reject if $\neg\mathsf{VerSoln}(\perp, str, puz, soln)$
13.			$\tau \leftarrow (N_C, N_S, M_1, M_1', puz, soln)$
14.			verify no stored presession $[\hat{C}, \hat{S}, \tau]$
15.			accept and store presession $[\hat{C}, \hat{S}, \tau]$
	continue with $\mathsf{CActionj}_P$		continue with $\mathsf{SActionj}_P$

Figure 1: $D(P)_{\mathsf{Puz},d,\mathsf{MAC},k}$ DoS countermeasure protocol.

of Send *queries issued and* t_0 *is a constant depending on the protocol, assuming* $t \in \mathrm{poly}(k)$.

The proof of Theorem 3 follows by a sequence of games, first replacing the message authentication code with a MAC challenger, and then replacing the puzzles with a Puz challenger. Fresh accepted presessions correspond to valid solutions to the Puz challenger, yielding the bound relating the protocol and the puzzle. The details appear in Appendix F.

6 Conclusion

Our goal in this work was to improve security definitions for client puzzles and denial-of-service-resistant protocols. We presented a new, stronger definition of puzzle difficulty for client puzzles, motivated by examples considering the effects of an adversary who has enough resources to solve more than one puzzle. This definition is sufficiently general to be useful for analyzing and proving the difficulty of a wide range of computation- and memory-bound puzzle constructions.

Whereas the client puzzle difficulty definition suffices for a simple game between a challenger and an adversary, we need something more advanced for a multi-user network setting. Thus, we introduced a new definition of DoS resistance for network protocols.

Our work can be viewed in part as combining the client puzzles approach of Chen et al. [CMSW09a] and the DoS-resistant protocols approach of Stebila and Ustaoglu, extending both to provide stronger DoS resistance and better modularity.

To demonstrate the utility of our new definitions, we have included examples of two hash-based client puzzles (including an analysis of the Hashcash client puzzle) and given a generic technique for converting any protocol into a DoS-resistant protocol using an interactive client puzzle.

Future Work. The interactive request-challenge-solution nature of client puzzles in the Chen et al. definition [CMSW09a] and our Definition 2 is incompatible with the definition of dynamic weakly verifiable puzzles [DIJK09], so the hardness amplification theorem from one to many puzzles does not apply. An important theoretical question arising is the development of a hardness amplification theorem for client puzzles that is suitable, and avoids the counterexamples from Sect. 2 when going from the Chen et al. definition [CMSW09a] to our Definition 3.2.1.

Key agreement is the most widely deployed cryptographic protocol on the Internet, and, as a computationally-expensive operation, is a possible attack vector for DoS attacks. Some Internet key agreement protocols – such as IKEv2 [Kau05], the Host Identity Protocol (HIP) [MNJH08], and Just Fast Keying (JFK) [ABB^{+}04] – have been designed with DoS attacks in mind. An important future work to be undertaken is the formal analysis of the DoS resistance of these protocols using an approach such as the one we have presented.

Acknowledgements

The authors are grateful for helpful suggestions from anonymous referees. This work was supported by the Australia-India Strategic Research Fund project TA020002.

References

[ABB^{+}04] William Aiello, Steven M. Bellovin, Matt Blaze, Ran Canetti, John Ioannidis, Angelos D. Keromytis, and Omer Reingold. Just Fast Keying: Key agreement in a hostile Internet. *ACM Transactions on Information and System Security*, **7**(2):1–30, May 2004. doi:10.1145/996943.996946.

[ABMW03] Mart n Abadi, Michael Burrows, Mark Manasse, and Ted Wobber. Moderately hard, memory-bound functions. In *Proc. Internet Society Network and Distributed System Security Symposium (NDSS) 2003*. Internet Society, 2003. url `http://www.isoc.org/isoc/conferences/ndss/03/proceedings/`.

[ADR02] Jee Hea An, Yevgeniy Dodis, and Tal Rabin. On the security of joint signature and encryption. In Lars Knudsen, editor, *Advances in Cryptology – Proc. EUROCRYPT 2002*, *LNCS*, volume 2332, pp. 83–107. Springer, 2002. doi:10.1007/3-540-46035-7_6.

[AN97] Tuomas Aura and Pekka Nikander. Stateless connections. In Yongei Han, Tatsuaki Okamoto, and Sihan Qing, editors, *Proc. 1st International Conference on Information and Communications Security (ICICS) 1997*, *LNCS*, volume 1334, pp. 87–97. Springer, 1997. doi:10.1007/BFb0028465.

[ANL00] Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. DOS-resistant authentication with client puzzles. In Bruce Christianson, Bruno Crispo, James A. Malcolm, and Michael Roe, editors, *Security Protocols: 8th International Workshop*, *LNCS*, volume 2133, pp. 170–177. Springer, 2000. doi:10.1007/3-540-44810-1_22.

[Bac97] Adam Back. A partial hash collision based postage scheme, 1997. url `http://www.hashcash.org/papers/announce.txt`.

[Bac04] Adam Back. Hashcash, 2004. url `http://www.hashcash.org/docs/hashcash.html#stamp_format__version_1_`.

[BCC^{+}08] Steve Babbage, Dario Catalano, Carlos Cid, Orr Dunkelman, Christian Gehrmann, Louis Granboulan, Tanja Lange, Arjen Lenstra, Phong Q. Nguyen, Christof Paar, Jan Pelzl, Thomas Pornin, Bart Preneel, Christian Rechberger, Vincent Rijmen, Matt Robshaw, Andy Rupp, Nigel Smart, and Michael Ward. ECRYPT yearly report on algorithms and keysizes (2007–2008), July 2008. url `http://www.ecrypt.eu.org/documents/D.SPA.28-1.1.pdf`.

[BKR00] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, **61**(3):362–399, 2000. DOI:10.1006/jcss.1999.1694.

[BM99a] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael Wiener, editor, *Advances in Cryptology – Proc. CRYPTO '99*, *LNCS*, volume 1666, pp. 786–786. Springer, 1999. DOI:10.1007/3-540-48405-1_28. Full version available as [BM99b].

[BM99b] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme, 1999. URL http://cseweb.ucsd.edu/users/mihir/papers/fsig.html. Short version published as [BM99a].

[Boy07] Xavier Boyen. Halting password puzzles: Hard-to-break encryption from human-memorable keys. In *Proc. 16th USENIX Security Symposium*, pp. 119–134, 2007. URL http://www.usenix.org/events/sec07/tech/boyen.html.

[BR93a] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *Advances in Cryptology – Proc. CRYPTO '93*, *LNCS*, volume 773, pp. 232–249. Springer, 1993. DOI:10.1007/3-540-48329-2_21. Full version available as [BR93b].

[BR93b] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution, 1993. URL http://www-cse.ucsd.edu/~mihir/papers/key-distribution.html. Extended abstract published as [BR93a].

[BR93c] Mihir Bellare and Phillip Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proc. 1st ACM Conference on Computer and Communications Security (CCS)*, pp. 62–73. ACM, 1993. DOI:10.1145/168588.168596.

[BS07a] Mihir Bellare and Sarah Shoup. Two-tier signatures, strongly unforgeable signatures, and fiat-shamir without random oracles. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *Public Key Cryptography (PKC) 2007*, *LNCS*, volume 4450, pp. 201–216. Springer, 2007. DOI:10.1007/978-3-540-71677-8_14. Full version available as [BS07b].

[BS07b] Mihir Bellare and Sarah Shoup. Two-tier signatures, strongly unforgeable signatures, and fiat-shamir without random oracles, 2007. URL http://cseweb.ucsd.edu/users/sshoup/pkc07.pdf. Extended abstract published as [BS07a].

[BSW06] Dan Boneh, Emily Shen, and Brent Waters. Strongly unforgeable signatures based on computational Diffie-Hellman. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography (PKC) 2006*, *LNCS*, volume 3958, pp. 229–240. Springer, 2006. DOI:10.1007/11745853_15.

[CHS05] Ran Canetti, Shai Halevi, and Michael Steiner. Hardness amplification of weakly verifiable puzzles. In Joe Kilian, editor, *Theory of Cryptography Conference (TCC) 2005*, *LNCS*, volume 3378, pp. 17–33. Springer, 2005. DOI:10.1007/b106171.

[CMSW09a] Liqun Chen, Paul Morrissey, Nigel P. Smart, and Bogdan Warinschi. Security notions and generic constructions for client puzzles. In Mitsuru Matsui, editor, *Advances in Cryptology – Proc. ASIACRYPT 2009*, *LNCS*, volume 5912, pp. 505–523. Springer, 2009. DOI:10.1007/978-3-642-10366-7_30. Full version available as [CMSW09b].

[CMSW09b] Liqun Chen, Paul Morrissey, Nigel P. Smart, and Bogdan Warinschi. Security notions and generic constructions for client puzzles, 2009. EPRINT http://eprint.iacr.org/2009/331. Short version published as [CMSW09a].

[DGN03a] Cynthia Dwork, Andrew Goldberg, and Moni Naor. On memory-bound functions for fighting spam. In Dan Boneh, editor, *Advances in Cryptology – Proc. CRYPTO 2003*, *LNCS*, volume 2729, pp. 426–444. Springer, 2003. DOI:10.1007/b11817. Full version available as [DGN03b].

[DGN03b] Cynthia Dwork, Andrew Goldberg, and Moni Naor. On memory-bound functions for fighting spam, 2003. URL http://www.wisdom.weizmann.ac.il/%7Enaor/PAPERS/mem_abs.html. Short version published as [DGN03a].

[DIJK09] Yevgeniy Dodis, Russell Impagliazzo, Ragesh Jaiswal, and Valentine Kabanets. Security amplification for interactive cryptographic primitives. In Omer Reingold, editor, *Theory of Cryptography Conference (TCC) 2009*, *LNCS*, volume 5444, pp. 128–145. Springer, 2009. DOI:10.1007/978-3-642-00457-5_9.

[DN92] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Advances in Cryptology – Proc. CRYPTO '92*, *LNCS*, volume 740, pp. 139–147. Springer, 1992. DOI:10.1007/3-540-48071-4_10.

[DNW05] Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and proofs of work. In Victor Shoup, editor, *Advances in Cryptology – Proc. CRYPTO 2005*, *LNCS*, volume 3621, pp. 37–54. Springer, 2005. DOI:10.1007/11535218_3.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, **17**(2):281–308, April 1988. DOI:10.1137/0217017.

[JB99] Ari Juels and John Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proc. Internet Society Network and Distributed System Security Symposium (NDSS) 1999*, pp. 151–165. Internet Society, 1999. URL http://www.isoc.org/isoc/conferences/ndss/99/proceedings/.

[JJ99] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols (extended abstract). In Bart Preneel, editor, *Proceedings of the IFIP TC6/TC11 Joint Working Conference on Secure Information Networks: Communications and Multimedia Security*, *IFIP Conference Proceedings*, volume 152, pp. 258–272. Kluwer, 1999. URL http://www.rsa.com/rsalabs/node.asp?id=2049.

[Jou04] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In Matt Franklin, editor, *Advances in Cryptology – Proc. CRYPTO 2004*, *LNCS*, volume 3152, pp. 306–316. Springer, 2004. DOI:10.1007/b99099.

[Kau05] Charlie Kaufman. Internet Key Exchange (IKEv2) protocol, December 2005. URL http://www.ietf.org/rfc/rfc4306.txt. RFC 4306.

[KC10] Ghassan O. Karame and Srdjan Capkun. Low-cost client puzzles based on modular exponentiation. In *Proc. 15th European Symposium on Research in Computer Security (ESORICS) 2010*, LNCS. Springer, 2010. To appear.

[LLM07] Brian LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *First International Conference on Provable Security (ProvSec) 2007*, *LNCS*, volume 4784, pp. 1–16. Springer, 2007. DOI:10.1007/978-3-540-75670-5_1.

[Mea99] Catherine Meadows. A formal framework and evaluation method for network denial of service. In *Proc. 12th IEEE Computer Security Foundations Workshop (CSFW) 1999*, p. 4. IEEE, 1999. DOI:10.1109/CSFW.1999.779758.

[MNJH08] Robert Moskowitz, Pekka Nikander, Petri Jokela, and Thomas R. Henderson. Host Identity Protocol, April 2008. URL http://www.ietf.org/rfc/rfc5201.txt. RFC 5201.

[MP02] Wenbo Mao and Kenneth G. Paterson. On the plausible deniability feature of Internet protocols. Manuscript, 2002. URL http://citeseer.ist.psu.edu/678290.html.

[MPM04] Timothy J. McNevin, Jung-Min Park, and Randolph Marchany. pTCP: A client puzzle protocol for defending against resource exhaustion denial of service attacks. Technical Report TR-ECE-04-10, Department of Electrical and Computer Engineering, Virginia Tech, October 2004. URL http://www.arias.ece.vt.edu/publications/TechReports/mcNevin-2004-1.pdf.

[RS97] Ronald L. Rivest and Adi Shamir. Payword and micromint: Two simple micropayment schemes. In *Security Protocols*, *LNCS*, volume 1189, pp. 69–87. Springer, 1997. DOI:10.1007/3-540-62494-5_6.

[RS04a] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Bimal Roy and Willi Meier, editors, *Proc. Fast Software Encryption (FSE) 2004*, *LNCS*, volume 3017, pp. 371–388. Springer, 2004. DOI:10.1007/b98177. Full version available as [RS04b].

[RS04b] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance, 2004. EPRINT http://eprint.iacr.org/2004/035. Extended abstract published as [RS04a].

[RSW96] Ronald L. Rivest, Adi Shamir, and David A. Wagner. Time-lock puzzles and timed-release crypto. Technical Report TR-684, MIT Laboratory for Computer Science, March 1996. URL http://people.csail.mit.edu/rivest/RivestShamirWagner-timelock.pdf.

[SGNB06] Jason Smith, Juan Gonz lez Nieto, and Colin Boyd. Modelling denial of service attacks on JFK with Meadows's cost-based framework. In Rajkumar Buyya, Tianchi Ma, Reihaneh Safavi-Naini, Chris Steketee, and Willy Susilo, editors, *Proc. 4th Australasian Information Security Workshop – Network Security (AISW-NetSec) 2006*, *CRPIT*, volume 54, pp. 125–134. Australian Computer Society, 2006. URL http://crpit.com/confpapers/CRPITV54Smith.pdf.

[Sho06] Victor Shoup. Sequences of games: A tool for taming complexity in security proofs, 2006. URL http://www.shoup.net/papers/games.pdf. First version appeared in 2004.

[Sti02] Douglas R. Stinson. *Cryptography: Theory and Practice*. Chapman & Hall, second edition, 2002.

[SU09] Douglas Stebila and Berkant Ustaoglu. Towards denial-of-service-resilient key agreement protocols. In Colin Boyd and Juan Gonz lez Nieto, editors, *Proc. 14th Australasian Conference on Information Security and Privacy (ACISP) 2009*, *LNCS*, volume 5594, pp. 389–406. Springer, 2009. DOI:10.1007/978-3-642-02620-1_27.

[TBFGN06] Suratose Tritilanunt, Colin Boyd, Ernest Foo, and Juan Gonz lez Nieto. Toward non-parallelizable client puzzles. In Feng Bao, San Ling, Tatsuaki Okamoto, Huaxiong Wang, and Chaoping Xing, editors, *Cryptology and Network Security (CANS) 2007*, *LNCS*, volume 4856. Springer, 2006. DOI:10.1007/978-3-540-76969-9_16.

[Ust08] Berkant Ustaoglu. Obtaining a secure and efficient key agreement protocol from (H)MQV and NAXOS. *Designs, Codes and Cryptography*, **46**(3):329–342, March 2008. DOI:10.1007/s10623-007-9159-1. EPRINT http://eprint.iacr.org/2007/123.

[WJHF04] Brent Waters, Ari Juels, J. Alex Halderman, and Edward W. Felten. New client puzzle outsourcing techniques for DoS resistance. In Birgit Pfitzmann and Peng Liu, editors, *Proc. 11th ACM Conference on Computer and Communications Security (CCS)*, pp. 246–256. ACM, 2004. DOI:10.1145/1030083.1030117.

A Background

A.1 Notation

We use different typefaces to denote *variables*, **constants**, algorithms, oracles, sets, and SECURITY NOTIONS. $a \leftarrow_R \mathsf{B}$ denotes a variable a being chosen uniformly at random from a set B. Throughout, "p.p.t." and "d.p.t." stand for probabilistic and deterministic polynomial time, respectively, d refers to a puzzle difficulty parameter, k refers to a security parameter, and s and ρ are secret keys. $\mathrm{negl}(k)$ denotes a negligible function in k, meaning it is asymptotically smaller than the inverse of any polynomial in k.

A.2 Message Authentication Codes

Definition 4 (Secure message authentication code [BKR00])**.** *A* family of keyed message authentication codes *is a set of functions* $\mathsf{MAC} : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k$*. Let $\mathcal{A}$ be a probabilistic algorithm. The experiment is the following algorithm:*

- $\mathsf{Exec}^{\text{EUCMA}}_{\mathsf{MAC},k}(\mathcal{A})$*: Set* $mk \leftarrow_R \{0,1\}^k$*. Set* $(m, \sigma) \leftarrow \mathcal{A}^{\mathsf{MAC}_{mk}(\cdot)}()$*. If* $\mathsf{MAC}_{mk}(m) = \sigma$ *and m was not queried to* $\mathsf{MAC}_{mk}(\cdot)$*, then return* **true***, otherwise return* **false***.*

Define

$$\mathsf{Adv}^{\text{EUCMA}}_{\mathsf{MAC},k}(q,t) = \max_{\mathcal{A}:q,t} \Pr\left(\mathsf{Exec}^{\text{EUCMA}}_{\mathsf{MAC},k}(\mathcal{A}) = \textbf{true}\right) \ ,$$

where the maximum is taken over all probabilistic algorithms $\mathcal{A}$ running in time at most t and making at most q queries to $\mathsf{MAC}_{mk}(\cdot)$ *in the experiment* $\mathsf{Exec}^{\text{EUCMA}}_{\mathsf{MAC},k}(\mathcal{A})$*. A family of message authentication codes* MAC *is* secure *if* $\mathsf{Adv}^{\text{EUCMA}}_{\mathsf{MAC},k}(q,t)$ *is a negligible function of k when q and t are polynomial in k.*

A.3 Client Puzzle Definition of Chen et al.

We now give a brief overview of the client puzzle and puzzle difficulty definitions of Chen et al. [CMSW09a]; they also include a puzzle unforgeability definition, but we omit this from our review as it is not relevant to our work (which we justify at the beginning of Sect. 3).

Definition 5 (Client Puzzle [CMSW09a])**.** *A* client puzzle *is a tuple of the following algorithms:*

- $\mathsf{Setup}(1^k)$*: Establishes parameter spaces, public parameters $params$, and long-term secret s.*
- $\mathsf{GenPuz}(s, d, str)$*: Generates a puzzle of difficulty d based on long-term secret s and string str.*
- $\mathsf{FindSoln}(puz, t)$*: Outputs a potential solution $soln$ for puzzle puz within running time t.*
- $\mathsf{VerAuth}(s, puz)$*: Checks the authenticity of puzzle puz using long-term secret s.*
- $\mathsf{VerSoln}(puz, soln)$*: Checks the correctness of solution $soln$ for puzzle puz.*

Definition 6 (Puzzle Difficulty [CMSW09a])**.** *Let d be a difficulty parameter and let k be a security parameter. Let $\mathcal{A}$ be a probabilistic algorithm with oracle access to oracles* $\mathsf{CreatePuzSoln}$ *and* Test*:*

- $\mathsf{CreatePuzSoln}(str)$*: Set* $puz \leftarrow \mathsf{GenPuz}(s, d, str)$ *and find a valid solution $soln$ for puz; return $(puz, soln)$.*
- $\mathsf{Test}(str)$*: Return* $puz \leftarrow \mathsf{GenPuz}(s, d, str)$*.*

Consider the following experiment for puzzle difficulty:

- $\mathsf{Exec}^{\text{DIFF}}_{\mathcal{A},\mathsf{Puz},d}(k)$*: Set* $(params, s) \leftarrow \mathsf{Setup}(1^k)$*. Run $\mathcal{A}$ with $params$; $\mathcal{A}$ is allowed to make any number of* $\mathsf{CreatePuzSoln}(str)$ *queries. At any point in time, $\mathcal{A}$ can make a single* $\mathsf{Test}(str)$ *query and receives puz. $\mathcal{A}$ outputs $soln$. Return* **true** *if* $\mathsf{VerSoln}(puz, soln) = \textbf{true}$*, and* **false** *otherwise.*

A client puzzle Puz *is said to be $\epsilon_{d,k}(\cdot)$-difficult if*

$$\mathsf{Adv}^{\text{DIFF}}_{\mathcal{A},\mathsf{Puz},d}(k) = \Pr\left(\mathsf{Exec}^{\text{DIFF}}_{\mathcal{A},\mathsf{Puz},d}(k) = \textbf{true}\right) \le \epsilon_{d,k}(t)$$

for all probabilistic algorithms $\mathcal{A}$ running in time at most t, where $\epsilon_{d,k}(t)$ is a family of functions monotonically increasing in t.

A.4 DoS Resistance Model of Stebila and Ustaoglu

We now give an overview of the definition of Stebila and Ustaoglu [SU09] for DoS resistance of key agreement protocols. A protocol consists of a presession – which contains a DoS countermeasure – followed by a session. An adversary interacts with a protocol by issuing Send queries (which deliver messages to parties) and DoSExpose($\hat{S}$) queries, which reveal a server's private information related to the DoS countermeasure. A key agreement protocol is denial-of-service resilient if:

1. when DoSExpose($\hat{B}$) has not been called, the presession identifiers $(\hat{A}, \hat{B}, ch, re)$ satisfy a *puzzling relation* (this models the difficulty of solving a puzzle), and
2. no expensive operations are performed by a server before a presession accepts.

A relation $\mathcal{R}$ is a *puzzling relation* if all of its members are tuples $(\hat{A}, \hat{B}, ch, re)$ for which it should be hard to produce a new solution re to a puzzle $(\hat{A}, \hat{B}, ch)$, even given access to an oracle that gives valid puzzle solutions. (This puzzle-solving oracle plays a similar to the role of the CreatePuzSoln query in the Chen et al. definition [CMSW09a].) It is important to note that, here, the adversary is allowed to query the puzzle-solving oracle on the target puzzle $(\hat{A}, \hat{B}, ch)$. In order to have broken the denial-of-service resistance of the protocol, a runtime-bounded adversary must simply return a "fresh" valid solution to $(\hat{A}, \hat{B}, ch)$, thereby causing a server $\hat{B}$ to accept a presession without having done the required work itself.

B Specification of MicroMint Counterexample Puzzle from Sect. 2

Puzzle Construction. In the model of Chen et al. [CMSW09a, Definition 1], a client puzzle is defined as a tuple of algorithms. Let $\mathsf{MAC} : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k$ be a family of keyed message authentication codes and let $H : \{0,1\}^* \to \{0,1\}^\ell$. The following algorithms define MMPuz, which is the puzzle from Sect. 2 using the MicroMint scheme, in the language of Chen et al. [CMSW09a] (as we describe in Appendix A.3).

- $\mathsf{Setup}(1^k)$: Let $\mathsf{sSpace} \leftarrow \{0,1\}^k$, $\mathsf{diffSpace} \leftarrow \{\ell/2\}$, $\mathsf{strSpace} \leftarrow \{0,1\}^*$, $\mathsf{puzSpace} \leftarrow \mathsf{strSpace} \times \{0,1\}^k \times \{0,1\}^k$, and $\mathsf{solnSpace} \leftarrow \{0,1\}^* \times \{0,1\}^*$. Set $mk \leftarrow_R \{0,1\}^\ell$. Set $params \leftarrow (\mathsf{sSpace}, \mathsf{puzSpace}, \mathsf{solnSpace}, \mathsf{diffSpace}, \bot)$. Return $(params, mk)$.
- $\mathsf{GenPuz}(mk, d = \ell/2, str)$: Set $m \leftarrow_R \{0,1\}^k$ and $\sigma \leftarrow \mathsf{MAC}_{mk}(str, m)$. Return (str, m, σ).
- $\mathsf{FindSoln}((str, m, \sigma), t)$: Find two values $x_1, x_2 \in \{0,1\}^*$ such that $H(str\|m\|x_1) = H(str\|m\|x_2)$ using a collision-finding algorithm (such as in [RS97, §4]). Return (x_1, x_2).
- $\mathsf{VerAuth}(mk, (str, m, \sigma))$: Return **true** if $\sigma = \mathsf{MAC}_{mk}(str, m)$ or **false** otherwise.
- $\mathsf{VerSoln}((str, m, \sigma), (x_1, x_2))$: Return **true** if $H(str\|m\|x_1) = H(str\|m\|x_2)$, or **false** otherwise.

It can be seen by inspection that this puzzle satisfies the correctness requirement of Chen et al. [CMSW09a].

Security Experiment. There are two security experiments defined by Chen et al.: unforgeability of puzzles and difficulty of solving puzzles. If MAC is a secure family of message authentication codes, then it is straightforward to show that the puzzle defined above satisfies the unforgeability requirement. We focus on the difficulty of solving puzzles since that is what motivates our work on powerful adversaries as described in Sect. 2.

For the purposes of the security experiment $\mathsf{Exec}^{\mathrm{DIFF}}_{\mathcal{A},\mathsf{MMPuz},d}(k)$, we define the algorithm

- $\mathsf{CreatePuzSoln}(str)$: Set $(str, m, \sigma) \leftarrow \mathsf{GenPuz}(mk, d, str)$ and $(x_1, x_2) \leftarrow \mathsf{FindSoln}((str, m, \sigma), \infty)$. Return $((str, m, \sigma), (x_1, x_2))$.

Fix $d = \ell/2$. We can see that $\mathcal{A}$ wins $\mathsf{Exec}^{\mathrm{DIFF}}_{\mathcal{A},\mathsf{MMPuz},d}(k)$ for $puz^\dagger = (str^\dagger, m^\dagger, \sigma^\dagger)$ with $soln^\dagger = (x_1^\dagger, x_2^\dagger)$ if and only if $H(str^\dagger\|m^\dagger\|x_1^\dagger) = H(str^\dagger\|m^\dagger\|x_2^\dagger)$. Assume $H(\cdot)$ is a random oracle; note that $H(str^\dagger\|m^\dagger\|\cdot)$ is also a random oracle. By the birthday paradox (c.f. [Sti02, §4.2.2]) the probability that an adversary $\mathcal{A}$ making q_H hash function queries can find a collision is approximately $q_H/2^{\ell/2}$, and thus

$$\mathrm{Adv}^{\text{DIFF}}_{\mathcal{A},\mathsf{MMPuz},d}(k) \leq \frac{q_H}{2^{\ell/2}} + \mathrm{negl}(k) \ . \tag{1}$$

This is a monotonically increasing function of q_H, so it satisfies the puzzle difficulty definition of Chen et al. [CMSW09a, Definition 3]. Moreover, it is of the form that one might expect for a reasonable client puzzle: approximately $t/2^d$, where t is the running time of the adversary and d is the difficulty of the puzzle. The random oracle-based puzzle from Chen et al. [CMSW09a, Appendix C] is of this form as well.

Remark. While the challenger must also make many hash function calls for each $\mathsf{CreatePuzSoln}$ query issued by the adversary, we do not need to account for these queries in (1) since the expression counts the number of queries made by the adversary, not the total running time of the adversary and challenger. If we were not working in the random oracle model, then we would need to make such an accounting.

Attack by Powerful Adversaries. By the birthday paradox, making approximately $\sqrt{n} \cdot 2^{\ell/2}$ hash function calls results in n hash function collisions. As a result, an adversary can solve n puzzles using only $\sqrt{n} \cdot 2^{\ell/2}$ queries, not $n \cdot 2^{\ell/2}$ queries as we would like.[7] We note that, using this attack, the n puzzle solutions will all be for the same challenge string $str^\dagger$, since we are working with random oracle $H(str^\dagger \| m^\dagger \| \cdot)$; however, the puzzle solutions will be distinct and thus the attack remains meaningful.

C Another counterexample puzzle based on signature forgery

In this appendix, we give an additional client puzzle counterexample in which n puzzles can be solved for less than n times the cost of solving one puzzle, similar to the MicroMint example in Section 2. In this puzzle, a legitimate client solves a puzzle by finding a signature forgery. The difficulty of the puzzle can be used to set the security parameter of the digital signature scheme. We typically want puzzles that are only moderately hard to solve – requiring, say, 2^{20} operations – which needs smaller signature scheme security parameters than usual. For the Chen et al. model [CMSW09a], we can use any signature scheme that is existentially unforgeable under chosen message attack [GMR88] suffices; for the Stebila and Ustaoglu model [SU09], we require a signature scheme that is *strongly* existentially unforgeable under chosen message attack [ADR02]. Interestingly, signature-forgery-based puzzle was first suggested in the paper by Dwork and Naor [DN92] that originally introduced client puzzles.

The basic idea is as follows. We employ a signature scheme by Bellare and Miner [BM99a]. To understand the puzzle construction, we focus on the verification algorithm:

- $\mathsf{Verify}(pk, m, (Y, Z))$: Set $c_1 \ldots c_\ell \leftarrow H(Y, m)$. Return **true** if $Z^2 = Y \cdot \prod_{i=1}^{\ell} U_i^{c_i} \mod N$, or **false** otherwise.

The important part is that an ℓ-bit hash is computed and then a test is performed to see if this hash satisfies a particular relation. For a client puzzle, one would typically choose ℓ sufficiently small, say, $\ell = 20$, so that a client could iterate through many possible $c_1 \ldots c_\ell$ to find one that satisfies the relation, requiring roughly $2^{\ell-1}$ iterations. This allows one to forge a single signature without obtaining the private keys.

An alternative method for forging signatures is to factor the RSA modulus N, obtain the private keys, and then use these to sign messages. The runtime of this procedure is dominated by the time required to factor the RSA modulus N.

[7] And if H is an iterated hash function, we only need to hash $\lceil \log_2 n \rceil 2^{\ell/2}$ values using Joux's multicollision attack [Jou04].

The cost of solving a single puzzle, then, is approximately $2^{\ell-1}$ operations. However, an adversary can solve n puzzles in time $t_{\mathsf{fact}}(k) + n \cdot t_{\mathsf{Sign}}$, where $t_{\mathsf{fact}}(k)$ is the time required to factor a k-bit RSA modulus and t_{Sign} is the time required for a signature. For sufficiently large n, this cost of solving n puzzles will be less than n times the cost of solving a single puzzle. For example, this is the case with $\ell = 20$, $k = 445$, $t_{\mathsf{fact}}(k) = 2^{46}$, $t_{\mathsf{Sign}} < 2^{\ell}$, and $n \geq 2^{27}$.

C.1 Background

Definition 7 (Signature scheme)**.** *A* signature scheme *is a tuple $\mathcal{S}$ of the following algorithms:*

- $\mathsf{KeyGen}(1^{\ell})$*: (p.p.t.) Returns public key pk and private key sk.*
- $\mathsf{Sign}(sk, m)$*: (p.p.t.) Returns σ, a signature of m under sk.*
- $\mathsf{Verify}(pk, m, \sigma)$*: (d.p.t.) Returns* **true** *or* **false***.*

Definition 8 (Existential unforgeability [GMR88])**.** *Let $\mathcal{S}$ be a signature scheme, $\mathcal{A}$ be a probabilistic algorithm, and $H : \{0,1\}^* \to \{0,1\}^{\ell}$ be a hash function. The experiment is the following algorithm:*

- $\mathsf{Exec}^{\text{UFCMA}}_{\mathcal{S},H}(\ell, \mathcal{A})$*: Set* $(pk, sk) \leftarrow \mathsf{KeyGen}(1^{\ell})$*. Set* $(m, \sigma) \leftarrow \mathcal{A}^{H, \mathsf{Sign}(sk, \cdot)}(pk)$*. If* $\mathsf{Verify}(pk, m, \sigma) =$ **true** *and m was not queried to* $\mathsf{Sign}(sk, \cdot)$*, return* **true***, otherwise return* **false***.*

Define

$$\mathrm{Adv}^{\text{UFCMA}}_{\mathcal{S},H}(\ell, t) = \max_{\mathcal{A}:t} \Pr\left(\mathsf{Exec}^{\text{UFCMA}}_{\mathcal{S},H}(\ell, \mathcal{A}) = \textbf{true}\right) \;\; ,$$

where the maximum is taken over all probabilistic algorithms $\mathcal{A}$ running in time at most t. A signature scheme $\mathcal{S}$ is said to be existentially unforgeable under a chosen message attack *if* $\mathrm{Adv}^{\text{UFCMA}}_{\mathcal{S},H}(\ell, t)$ *is a negligible function of ℓ for all $t \in \mathrm{poly}(\ell)$.*

Bellare-Miner signature scheme. The Bellare-Miner signature scheme (without forward security) $\mathcal{S}_{\mathbf{BM}}$ is defined as the tuple consisting of the following three algorithms [BM99a]. [8]

- $\mathsf{KeyGen}(1^{\ell})$: Let $k = k(\ell)$ be the size of an RSA modulus required for security parameter ℓ. Pick random distinct $k/2$-bit primes $p, q \equiv 3 \mod 4$. Set $N \leftarrow pq$. For $i = 1, \ldots, \ell$, set $S_i \leftarrow_R \mathbb{Z}_N^*, U_i \leftarrow S_i^4 \mod N$. Return public key $pk = (N, U_1, \ldots, U_{\ell})$, private key $sk = (N, S_1^2, \ldots, S_{\ell}^2)$.
- $\mathsf{Sign}(sk, m)$: Pick $R \leftarrow_R \mathbb{Z}_N^*$. Set $Y \leftarrow R^2 \mod N$, $c_1 \ldots c_{\ell} \leftarrow H(Y, m)$, $Z \leftarrow R \cdot \prod_{i=1}^{\ell} S_i^{2c_i} \mod N$. Return (Y, Z).
- $\mathsf{Verify}(pk, m, (Y, Z))$: Set $c_1 \ldots c_{\ell} \leftarrow H(Y, m)$. Return **true** if $Z^2 = Y \cdot \prod_{i=1}^{\ell} U_i^{c_i} \mod N$, or **false** otherwise.

As a corollary of Theorem 4.2 of Bellare and Miner [BM99a], we have that, for an adversary running in time t usingat most q_H calls to H and at most q_{Sign} calls to Sign,

$$\mathrm{Adv}^{\text{UFCMA}}_{\mathcal{S}_{\mathbf{BM}},H}(\ell, t, q_H, q_{\mathsf{Sign}}) \leq 2q_H\left(2^{-\ell} + \sqrt{2\ell \mathrm{Adv}^{\text{FACT}}(k, t')}\right) + 2^{2-k} q_H q_{\mathsf{Sign}} \;\; , \tag{2}$$

where $t' = 2t + O(k^3)$ and $\mathrm{Adv}^{\text{FACT}}(k, t)$ is the probability that a probabilistic algorithm running in time t can factor a k-bit RSA modulus $N = pq$ where $p, q \equiv 3 \mod 4$ are random distinct $k/2$-bit primes. We take $\mathrm{Adv}^{\text{FACT}}(k, t) \leq t / t_{\mathsf{fact}}(k)$, where $t_{\mathsf{fact}}(k) = 2^{s(k)}$ and $s(k) = \left(\frac{64}{9}\right)^{1/3} \log_2(e)(k \ln 2)^{1/3}(\ln(k \ln 2))^{2/3} - 14$ [BCC+08, §6.2].

[8] In fact, the Bellare-Miner scheme is a forward secure signature scheme, in which signing keys can evolve over time and the compromise of the current signing key does not allow forging of signatures for previous time periods. We do not require this property, but fortunately the security definition reduces to the standard one when the number of time periods is taken to be 1, which we assume in the rest of the paper. Our main motivation for choosing this scheme as the basis of our puzzle is that it is a Feige-Fiat-Shamir-like signature scheme with a proof of security.

C.2 In the Chen et al. model

In this section, we specify the puzzle construction fully in the model of of Chen et al. [CMSW09a] and prove that it satisfies their definition of a difficult puzzle.

Puzzle construction. Let $\mathcal{S}_{\mathbf{BM}}$ be the Bellare-Miner signature scheme C.1. Define the following algorithm:

- $\mathsf{SBMForge}(pk, m)$: Set $c_1 \dots c_\ell \leftarrow_R \{0,1\}^\ell$ and $U \leftarrow \prod_{i=1}^{\ell} U_i^{c_i} \mod N$. Repeat the following: set $Z \leftarrow_R \mathbb{Z}_N^*, Y \leftarrow Z^2/U \mod N$; until $H(Y, m) = c$. Return (Y, Z).

In the model of Chen et al. [CMSW09a, Definition 1], a client puzzle is defined as a tuple of algorithms. Let $\mathsf{MAC} : \{0,1\}^\ell \times \{0,1\}^* \to \{0,1\}^\ell$ be a family of keyed message authentication codes. The following algorithms define SBMPuz, which is the puzzle from Section 2 using the Bellare-Miner signature scheme in the language of Chen et al. [CMSW09a] (as we describe in Appendix A.3).

- $\mathsf{Setup}(1^\ell)$: Set $(pk, sk) \leftarrow \mathsf{KeyGen}(1^\ell)$, $pk \leftarrow (N, U_1, \dots, U_\ell)$, and $sk \leftarrow (N, S_1^2, \dots, S_\ell^2)$. Let $\mathsf{sSpace} \leftarrow \{0,1\}^\ell$, $\mathsf{diffSpace} \leftarrow \{2^\ell\}$, $\mathsf{strSpace} \leftarrow \{0,1\}^*$, $\mathsf{puzSpace} \leftarrow \mathsf{strSpace} \times \{0,1\}^\ell \times \{0,1\}^\ell$, and $\mathsf{solnSpace} \leftarrow \mathbb{Z}_N^* \times \mathbb{Z}_N^*$. Set $mk \leftarrow_R \{0,1\}^\ell$. Set $\Pi \leftarrow pk$ and $params \leftarrow (\mathsf{sSpace}, \mathsf{puzSpace}, \mathsf{solnSpace}, \mathsf{diffSpace}, \Pi)$. Return $(params, (sk, mk))$.
- $\mathsf{GenPuz}((sk, mk), d, str)$: Set $m \leftarrow_R \{0,1\}^\ell$ and $\sigma \leftarrow \mathsf{MAC}_{mk}(str, m)$. Return (str, m, σ).
- $\mathsf{FindSoln}((str, m, \sigma), t)$: Run $\mathsf{SBMForge}(pk, m)$ until time t has elapsed or it returns a value (Y, Z), whichever comes first. Return (Y, Z).
- $\mathsf{VerAuth}((sk, mk), (str, m, \sigma))$: Return **true** if $\sigma = \mathsf{MAC}_{mk}(str, m)$ or **false** otherwise.
- $\mathsf{VerSoln}((str, m, \sigma), (Y, Z))$: Set $c_1 \dots c_\ell \leftarrow H(Y, m)$. Return **true** if $Z^2 \equiv Y \cdot \prod_{i=1}^{\ell} U_i^{c_i} \mod N$, or **false** otherwise.

It can be seen by inspection that this puzzle satisfies the correctness requirement of Chen et al. [CMSW09a].

Security experiment. There are two security experiments defined by Chen et al.: unforgeability of puzzles and difficulty of solving puzzles. If MAC is a secure family of message authentication codes, then it is straightforward to show that the puzzle defined above satisfies the unforgeability requirement. We focus on the difficulty of solving puzzles since that is what motivates our work on powerful adversaries as described in Section 2.

For the purposes of the security experiment $\mathsf{Exec}^{\text{DIFF}}_{\mathcal{A},\mathsf{SBMPuz},d}(\ell)$, we define the algorithm

- $\mathsf{CreatePuzSoln}(str)$: Set $(str, m, t) \leftarrow \mathsf{GenPuz}((sk, mk), d, str)$ and $(Y, Z) \leftarrow \mathsf{Sign}(sk, m)$. Return $((str, m, t), (Y, Z))$.

Fix $d = 2^\ell$. We can see that $\mathcal{A}$ wins $\mathsf{Exec}^{\text{DIFF}}_{\mathcal{A},\mathsf{SBMPuz},d}(\ell)$ for $puz^\dagger = (str^\dagger, m^\dagger, t^\dagger)$ with $soln^\dagger = (Y^\dagger, Z^\dagger)$ if and only if $(Y^\dagger, Z^\dagger)$ is a signature of $m^\dagger$ in the $\mathcal{S}_{\mathbf{BM}}$ signature scheme. Moreover, since every m in every puzzle generated by CreatePuzSoln was generated randomly, as was $m^\dagger$ in the call to $\mathsf{GenPuz}(s, d, str^\dagger)$, we have that, except with negligible (in ℓ) probability, $m^\dagger$ was never an input to $\mathsf{Sign}(sk, \cdot)$. Thus, $(Y^\dagger, Z^\dagger)$ is a forgery for $m^\dagger$ under chosen message attack. For an adversary $\mathcal{A}$ running in time at most t, we therefore have

$$\begin{aligned} \mathrm{Adv}^{\text{DIFF}}_{\mathcal{A},\mathsf{SBMPuz},d}(\ell) &\le \mathrm{Adv}^{\text{UFCMA}}_{\mathcal{S}_{\mathbf{BM}},H}\left(\ell, t + O(q_C), q_H, q_C\right) + \mathrm{negl}(\ell) \\ &\le 2q_H 2^{-\ell} + 2q_H \sqrt{2\ell \mathrm{Adv}^{\text{FACT}}(k, 2t + O(q_C) + k^3)} + 2^{2-k} q_H q_C + \mathrm{negl}(\ell) \ , \end{aligned} \tag{3}$$

where q_C is the number of CreatePuzSoln queries issued and q_H is the number of hash function queries issued by $\mathcal{A}$, and k is the size of the RSA modulus. This is a monotonically increasing function of t, so it satisfies the puzzle difficulty definition of Chen et al. [CMSW09a, Definition 3]. Moreover, it is of the form that one might expect for a reasonable client puzzle: approximately t/d, where t is the running time of the adversary and d is the difficulty of the puzzle. The random oracle-based puzzle from Chen et al. [CMSW09a, Appendix C] is of this form as well.

We want the expression in equation (3) to be approximately bounded by $q_H/2^{\ell-1} + \mathrm{negl}(\ell)$. To do so requires each term in equation (3) to be (approximately) less than or equal to $q_H/2^{\ell-1} + \mathrm{negl}(\ell)$. For the first term, it follows vacuously. For the second term, it is satisfied when $s(k) \geq 1 + \log_2 \ell + 2\log_2 t$, where $s(k)$ is as defined in Appendix C.1.

Attack by powerful adversaries. For an example that allows an attack by a powerful adversary, we will choose the difficulty parameter ℓ and the size of the RSA modulus $k = k(\ell)$ such that both forging procedures satisfy the puzzle difficulty requirement. However, an adversary who uses the factoring approach will be able to solve many puzzles for less than the cost of solving those puzzles individually using the algorithm $\mathsf{SBMForge}$. In particular, an adversary using the factoring approach can solve n puzzles in time roughly $t_{\mathsf{fact}}(k) + nt_{\mathsf{Sign}}$, whereas the cost to solve n puzzles individually should be $n2^{\ell-1}$. When $\ell = 20$, equation (2) requires $s(k) \geq 46$ and hence $k \geq 445$. This means that an adversary can solve n puzzles in time $2^{46} + nt_{\mathsf{Sign}}$, which, when $t_{\mathsf{Sign}} < 2^{20}$, is less than $n \cdot 2^{20}$ for $n \geq 2^{27}$.

C.3 In the Stebila and Ustaoglu model

For the Stebila and Ustaoglu model, a puzzle construction using digital signatures would require a *strongly* unforgeable signature scheme [ADR02]. A few such schemes exist [ADR02, BSW06, BS07a] and could be used generically to construct a DoS countermeasure in which the solution to a puzzle is a forgery of a signature on the puzzle. This would result in a countermeasure that satisfies the Stebila and Ustaoglu definition, but for which a powerful adversary can solve n puzzles for less than n times the cost of solving a single puzzle.

C.4 In this paper's model

This signature-forgery-based puzzle that we constructed to demonstrate a weakness in existing puzzle difficulty definitions does not satisfy our definition of strong puzzle difficulty from Section 3.2. Recall that there are two strategies to solve the signature-forgery-based puzzles: (1) construct a forgery on a single message by finding a preimage in the hash function, or (2) factor the RSA modulus to recovery the signing key and then sign the message. Suppose we have chosen our system so that the amount of time it takes to perform (1) is $d = 2^{20}$ operations, the time it takes to factor the RSA modulus is $2^k = 2^{46}$ operations, and the time it takes to sign a message is $t_{\mathsf{Sign}} < d = 2^{20}$. Suppose $\mathcal{A}$ is asked to solve n puzzles in time t. If $t < 2^{46} + n \cdot t_{\mathsf{Sign}}$, then $\mathcal{A}$ employs strategy (1), otherwise it employs strategy (2). Therefore,

$$\Pr\left(\mathsf{Exec}^{\text{INT-STR-DIFF}}_{\mathcal{A},n,2^{20},\mathsf{SBMPuz}}(46) = \mathbf{true}\right) \geq \min\left\{1, f(t,n)\right\} \ .$$

where

$$f(t,n) = \max\left\{\frac{t}{n2^{20}}, \frac{t}{2^{46} + n \cdot t_{\mathsf{Sign}}}\right\} \ .$$

We observe that $f(t,n) > f(t/n,1)$ for $n > 2^{27}$ since $t_{\mathsf{Sign}} < d = 2^{20}$:

$$f(t,n) = \frac{t}{2^{46} + nt_{\mathsf{Sign}}} \approx \frac{t}{nt_{\mathsf{Sign}}} > \frac{t}{2^{20}n} = \frac{t/n}{2^{20}} = f(t/n,1) \ .$$

As a result, the signature-forgery-based puzzle is not an $\epsilon_{d,k,n}(\cdot)$-strongly difficult interactive puzzle according to Definition 2 for $\epsilon_{d,k,n}(t) = t/dn + \mathrm{negl}(k)$ (or any constant multiple thereof).

D Difficulty of the SPuz Hash Function Inversion Client Puzzle

In this appendix, we present a proof that the **SPuz** hash function inversion client puzzle from Sect. 3.3 is a strongly-difficult interactive client puzzle in the random oracle model [BR93c].

Theorem 1. *Let H be a random oracle. Let $\epsilon_{d,k,n}(q) \approx \left(\frac{q+n}{n2^d}\right)^n$. Then $\mathbf{SPuz}_H$ is an $\epsilon_{d,k,n}(q)$-strongly-difficult interactive client puzzle, where q is the number of distinct queries to H.*

Proof. For the $\mathsf{Exec}^{\text{INT-STR-DIFF}}$ experiment, we need to specify how the GetSoln oracle obtains a solution to a generated puzzle.

- $\mathsf{GetSoln}(str, (x'', y))$: If (x'', y) was recorded by GetPuz, then return x', where $x = x' \| x''$ was the random bit string chosen in GenPuz for this puzzle; otherwise, return $\perp$.

The proof proceeds using a counting argument. Fix d. Let $\mathcal{A}$ be a probabilistic algorithm. Clearly, there is a strategy for $\mathcal{A}$ to win the experiment with probability 1, by making at most $n2^d$ calls to H: for each of n puzzles $puz_i = (x_i'', y_i)$, try all strings z of length d until one is found such that $H(z \| x_i'', d, str_i) = y_i$. In the random oracle model, this is essentially the optimal strategy.

Let $Z_i = \{z_{i,1}, \ldots, z_{i,q_i+1}\} \subseteq \{0,1\}^*$ with $|Z_i| = q_i + 1$ (that is, the set contains no repetitions). Let $E_{i,j}$ be the event that $H(z_{i,j}) = y_i$ for $j = 1, \ldots, q_i$. Since the output of H is independent and uniformly random, and since x_i was chosen independently and uniformly at random, we have that $\Pr(E_{i,j}) \leq 2^{-d}$. Let F_i be the event that there exists $z_{i,j} \in Z_i$ such that $H(z_{i,j}) = y_i$; in other words, $F_i = \bigvee_{j=1}^{q_i+1} E_{i,j}$. Let $q_i \in \{0, \ldots, d\}$ be the number of queries issued to H for puzzle $i = 1, \ldots, n$, so that $q = \sum_{i=1}^n q_i$. Then

$$\begin{aligned}
\Pr\left(\bigwedge_{i=1}^n F_i\right) &= \prod_{i=1}^n \Pr(F_i) &&= \prod_{i=1}^n \Pr\left(\bigvee_{j=1}^{q_i+1} E_{i,j}\right) &&\leq \prod_{i=1}^n \sum_{j=1}^{q_i+1} \Pr(E_{i,j}) \\
&= \prod_{i=1}^n \frac{q_i+1}{2^d} &&\leq \left(\frac{\sum_{i=1}^n (q_i+1)}{n2^d}\right)^n = \left(\frac{q+n}{n2^d}\right)^n
\end{aligned}$$

We note that any adversary making q_i queries to H has at best a probability of $(q_i+1)/2^d$ of returning a value z_i that satisfies $H(z_i) = y_i$: checking q_i values using H, and, if that fails then guessing at random one of the remaining values. Thus,

$$\Pr\left(\mathsf{Exec}^{\text{INT-STR-DIFF}}_{\mathcal{A},n,d,\mathbf{SPuz}}(k) = \mathbf{true}\right) \leq \Pr\left(\bigwedge_{i=1}^n F_i\right) \lessapprox \left(\frac{q+n}{n2^d}\right)^n = \epsilon_{d,k,n}(q) \ .$$

Finally it is easy to see that $\epsilon_{2^d,k,n}(q) \leq \epsilon_{2^d,k,1}(q/n)$ for all n and q such that $\epsilon_{d^d,k,n}(q) \leq 1$.

Thus, **SPuz** is an $\epsilon_{2^d,k,n}(q)$-strongly-difficult interactive client puzzle. □

E Specification and Difficulty of the Hashcash Client Puzzle

In this appendix, we specify Hashcash as a client puzzle in the language of Sect. 3.1 and prove that it is a strongly-difficult non-interactive client puzzle.

A Hashcash *stamp* [Bac04] is a string of the form

```
ver:bits:date:resource:[ext]:rand:counter
```

- `ver` is a version identifier and is fixed to 1;
- `bits` is the "value" of the stamp: the number of zeros at the start of the output;
- `date` is the date the stamp is intended for, in the format `YYMMDD[hhmm[ss]]`;

- `resource` is the name of the resource this stamp is intended for; for email, this is the recipient's email address, such as `test@example.com`;
- `ext` (optional) is reserved for future extensions and is presently not supported;
- `rand` is a random string in Base64*, chosen by the client to avoid collisions with other senders' stamps;
- `counter` is a string in Base64* that is the solution to the puzzle.

Here, $\mathsf{Base64} = \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9, +, /, =\}$.

Let $H : \{0,1\}^* \to \{0,1\}^k$ be a hash function. Define $\mathbf{Hashcash}_H$ to be the following tuple of algorithms:

- $\mathsf{Setup}(1^k)$: Set $\mathsf{sSpace} \leftarrow \{\perp\}$, $\mathsf{diffSpace} \leftarrow \{0,1,\ldots\}$, $\mathsf{strSpace}$ to be the set of strings of the form `ver:bits:date:resource` where `ver` $= 1$, `bits` $\in \mathsf{diffSpace}$, `date` is a correctly-formatted date as specified above, and `resource` is a correctly-formatted email address, $\mathsf{puzSpace} \leftarrow \mathsf{strSpace} \times \mathsf{Base64}^*$, and $\mathsf{solnSpace} \leftarrow \mathsf{Base64}^*$. Set $s \leftarrow \perp$.
- $\mathsf{GenPuz}(\perp, d, str)$: Set `rand` $\leftarrow_R \mathsf{Base64}^k$ and $puz \leftarrow str$ `:rand`.
- $\mathsf{FindSoln}(str, puz, t)$: For i from 0 to $\max\{t, 2^d\}$: set $soln \leftarrow i$ (in Base64); if $H(puz{:}soln)_{[1\ldots d]} = 0\ldots0$ then return $soln$.
- $\mathsf{VerSoln}(\perp, str, puz, soln)$, where $puz =$ `ver:bits:date:resource::rand`: If any of the following checks fail, return **false**: check `ver` $= 1$, `bits` $\geq d$, `date` is valid, `resource` is valid (for the application), $puz{:}soln$ is not in the double-spend database, and $H(puz{:}soln)_{[1\ldots\texttt{bits}]} = 0\ldots0$. Store $puz{:}soln$ in the double-spend database. Return **true**.

Theorem 2. *Let $H : \{0,1\}^* \to \{0,1\}^k$, $k \geq d$, be a random oracle. Let $\epsilon_{d,k,n}(q) = \frac{q+n}{n2^d}$. Then $\mathbf{Hashcash}_H$ is an $\epsilon_{d,k,n}(q)$-strongly-difficult non-interactive client puzzle, where q is the number of distinct queries made by $\mathcal{A}$ to H.*

Remark. Theorem 2 assumes that the hash function H behaves as a random oracle. It would be desirable to use a more concrete hash function property [RS04a], such as preimage resistance. However, the non-interactive nature of the definition seems to preclude such a security reduction.

Proof. Fix d. Let $x_1, \ldots, x_q$ be the q distinct queries issued by $\mathcal{A}$ to H. Note that, although there is no shortcut for the challenger to respond to $\mathsf{CreatePuzSoln}$ queries, we only need to count $\mathcal{A}$'s queries to H since H is a random oracle. Let E_i be the binary random variable corresponding to the event that x_i is a valid Hashcash stamp — in other words, $E_i = 1$ if and only if $H(x_i)_{[1\ldots d]} = 0\ldots0$. Then $\Pr(E_i = 1) \leq 1/2^d$. Since H is a random oracle and the x_i are distinct, the E_i are independent.

Let $F = \sum_{i=1}^{q} E_i$. By Markov's inequality, $\Pr(F \geq n) \leq E(F)/n$. Since F is a binomial random variable with parameters q and $p \leq \frac{1}{2^d}$, we have that $E(F) \leq q/2^d$, and hence $\Pr(F \geq n) \leq q/n2^d$.

The best strategy for an adversary is to make q queries to H to attempt to find valid Hashcash stamps, and then, if it has found less than n such stamps, to return random guesses (without checking the oracle) for any remaining stamps; there are at most n such guesses made. Hence,

$$\Pr\left(\mathsf{Exec}^{\text{NINT-STR-DIFF}}_{\mathcal{A},n,d,\mathbf{Hashcash}_H}(k) = \mathbf{true}\right) \leq \frac{q+n}{n2^d} = \epsilon_{d,k,n}(q) \ .$$

Finally, we note that $\epsilon_{d,k,n}(q) = \frac{q+n}{n2^d} = \epsilon_{d,k,1}(q/n)$ and thereby satisfies $\epsilon_{d,k,n}(q) \leq \epsilon_{d,k,1}(q/n)$. Thus, $\mathbf{Hashcash}_H$ is an $\epsilon_{d,k,n}(q)$-strongly-difficult non-interactive client puzzle. □

F Denial of Service Resistance of $D(P)_{\mathsf{Puz},d,\mathsf{MAC},k}$

In this section, we give a proof of Theorem 3, that $D(P)_{\mathsf{Puz},d,\mathsf{MAC},k}$ is a DoS-resistant protocol.

Theorem 3. *Let P be a protocol such that $\mathsf{SAction1}_P$ does not involve any expensive operations. Suppose that Puz is an $\epsilon_{d,k,n}(t)$-strongly-difficult interactive puzzle with long-term secret key space $\mathsf{sSpace} = \{\perp\}$ and that MAC is a family of secure message authentication codes. Then $D(P)_{\mathsf{Puz},d,\mathsf{MAC},k}$ is an $\epsilon'_{d,k,n}(t)$-denial-of-service-resistant protocol, for $\epsilon'_{k,n}(t) = \epsilon_{d,k,n}(t + t_0 q_{\mathsf{Send}}) + \mathrm{negl}(k)$, where q_{Send} is the number of Send queries issued and t_0 is a constant depending on the protocol, assuming $t \in \mathrm{poly}(k)$.*

Proof. The argument for the first part of Definition 3 proceeds using a sequence of games [Sho06]. The proof idea is relatively straightforward. First, using a hybrid argument in games $G_1, \ldots, G_m$, we replace the message authentication code MAC with a private list of authenticated messages and reject any messages not on that list; if an adversary can distinguish this from the original protocol, then we have a forging algorithm for MAC. Next, in game G_{m+1}, we use the fresh accepted presessions in the protocol as solutions to the strongly-difficult interactive client puzzle game.

Game G_0. Let G_0 denote the original protocol $D(P)_{\mathsf{Puz},d,\mathsf{MAC},k}$. Then

$$\Pr\left(\mathsf{Exec}^{\text{DOS}}_{\mathcal{A},n,D(P)_{\mathsf{Puz},d,\mathsf{MAC},k}}(k) = \mathbf{true}\right) = \Pr\left(\mathsf{Exec}^{\text{DOS}}_{\mathcal{A},n,G_0}(k) = \mathbf{true}\right) \ . \tag{4}$$

Game G_i, for $i = 1, \ldots, |\mathsf{Servers}|$. We now describe a sequence of games $G_1, \ldots, G_m$, where $m = |\mathsf{Servers}|$. First, let $\hat{S}^*_1, \ldots, \hat{S}^*_m$ be a random permutation of $\mathsf{Servers}$. Let $\mathsf{Servers}^*_i = \{\hat{S}^*_1, \ldots, \hat{S}^*_{i-1}\}$, and let $\overline{\mathsf{Servers}}^*_i = \{\hat{S}^*_{i+1}, \ldots, \hat{S}^*_m\}$. Initialize the experiment $\mathsf{Exec}^{\text{EUCMA}}_{\mathsf{MAC},k}$ with oracle $\mathsf{MAC}^*(\cdot)$.

We define game G_i, for $i = 1 \ldots, m$, as the same game as G_{i-1} with the following modifications:

- $\mathsf{ServerSetup}(\hat{S})$: If $\hat{S} \neq \hat{S}^*_i$, then set $mk_{\hat{S}} \leftarrow_R \{0,1\}^k$ and $\rho_{\hat{S}} \leftarrow mk_{\hat{S}}$, otherwise set $\rho_{\hat{S}} \leftarrow \perp$.
- $\mathsf{Expose}(\hat{S})$: If $\hat{S} \neq \hat{S}^*_i$, then return $mk_{\hat{S}}$, otherwise abort.
- $\mathsf{Send}(\hat{S}, i, M)$: The following lines from Figure 1 are replaced:
  7. If $\hat{S} \neq \hat{S}^*_i$, then $\sigma \leftarrow \mathsf{MAC}_{mk_{\hat{S}}}(str, puz)$; otherwise, set $\sigma \leftarrow \mathsf{MAC}^*(str, puz)$. If $\hat{S} \in \mathsf{Servers}^*_i \cup \hat{S}^*_i$, then add (str, puz, σ) to $MACList_{\hat{S}}$.
  11. If $\hat{S} \in \mathsf{Servers}^*_i$ and $(str, puz, \sigma) \notin MACList_{\hat{S}}$, then reject; else if $\hat{S} = \hat{S}^*_i$ and $(str, puz, \sigma) \notin MACList_{\hat{S}}$, then try $((str, puz), \sigma)$ as a MAC^* forgery; else if $\hat{S} \in \overline{\mathsf{Servers}}^*_i$, then reject if $\sigma \neq \mathsf{MAC}_{mk_{\hat{S}}}(str, puz)$.

Let E_i be the event that $\mathsf{Expose}(\hat{S}^*_i)$ is not called and $\hat{S}^*_i$ receives a message in line 11 that is a valid MAC tag but is not in $MACList_{\hat{S}^*_i}$. Then, when E_i does not occur, any adversary that can distinguish the probability distribution of messages presented in game G_{i-1} from the distribution of messages presented in game G_i can be used as a MAC^* forger. Hence,

$$\left|\Pr\left(\mathsf{Exec}^{\text{DOS}}_{\mathcal{A},n,G_{i-1}}(k) = \mathbf{true}\right) - \Pr\left(\mathsf{Exec}^{\text{DOS}}_{\mathcal{A},n,G_i}(k) = \mathbf{true}\right)\right| \leq \Pr(E_i)\mathsf{Adv}^{\text{EUCMA}}_{\mathsf{MAC},k}(q_{\mathsf{Send}}, t) \ , \tag{5}$$

where q_{Send} is the number of Send queries issued in game G_i and t is the running time of $\mathcal{A}$ plus a constant multiple of q_{Send}. Note that since a valid adversary $\mathcal{A}$ against $\mathsf{Exec}^{\text{DOS}}_{\mathcal{A},n,G_i}$ must leave at least one server unexposed, we have that $\Pr(E_i) \geq 1/|\mathsf{Servers}|$.

Game G_{m+1}. Initialize the experiment $\mathsf{Exec}^{\text{INT-STR-DIFF}}_{G_{m+1},n\mathsf{Puz},d}(k)$ with oracles $\mathsf{GetPuz}(\cdot)$ and $\mathsf{GetSoln}(\cdot,\cdot)$. We define game G_{m+1} as the same game as G_0 with the following modifications:

- $\mathsf{Send}(\hat{S}, i, M)$: The following lines from Figure 1 are replaced:

6. $puz \leftarrow \mathsf{GetPuz}(str)$.
7. $\sigma \leftarrow \mathsf{MAC}_{mk_{\hat{S}}}(str, puz)$; add (str, puz, σ) to $MACList_{\hat{S}}$.
9. $soln \leftarrow \mathsf{GetSoln}(str, puz)$.
11. Reject if $(str, puz, \sigma) \notin MACList_{\hat{S}}$.

Let $\{(\hat{C}_i, \hat{S}_i, \tau_i)\}$, where $\tau_i = (N_i, N_i', \ldots_i, puz_i, soln_i)$, be the set of n fresh presessions accepted in G_{m+1}. Then return the set $\{(str_i, puz_i, soln_i)\}$, where $str_i = (\hat{C}_i, \hat{S}_i, N_i, N_i', \ldots_i)$, as the set of puzzle solutions in $\mathsf{Exec}^{\text{INT-STR-DIFF}}_{G_{m+1},n,\mathsf{Puz},d}(k)$.

First, we note that the probability distributions of messages in games G_m and G_{m+1} are identical, so

$$\Pr\left(\mathsf{Exec}^{\text{DOS}}_{\mathcal{A},n,G_m}(k) = \textbf{true}\right) = \Pr\left(\mathsf{Exec}^{\text{DOS}}_{\mathcal{A},n,G_{m+1}}(k) = \textbf{true}\right) \ . \tag{6}$$

Next, we need to compute $\Pr\left(\mathsf{Exec}^{\text{DOS}}_{\mathcal{A},n,G_{m+1}}(k) = \textbf{true}\right)$. By the construction of G_{m+1}, this happens precisely when $\mathsf{Exec}^{\text{INT-STR-DIFF}}_{G_{m+1},n,\mathsf{Puz},d}(k) = \textbf{true}$, since all puzzle-solution pairs received by G_{m+1} correspond to puzzles generated by GetPuz. Hence,

$$\Pr\left(\mathsf{Exec}^{\text{DOS}}_{A,n,G_{m+1}}(k) = \textbf{true}\right) \leq \Pr\left(\mathsf{Exec}^{\text{INT-STR-DIFF}}_{G_{m+1},n,\mathsf{Puz},d}(k) = \textbf{true}\right) \ .$$

Since each presession in question is fresh, there is no client instance with a matching conversation. In other words, there was no query to $\mathsf{GetSoln}$ with str, puz. Thus, if the solution is valid in the presession, then it is also valid for the Puz challenger.

Let t be the running time of $\mathcal{A}$, let $t_0 q_{\mathsf{Send}}$ be the running time of G_{m+1} excluding $\mathcal{A}$ (where t_0 is some constant specified by the protocol), and let $t' = t + t_0 q_{\mathsf{Send}}$. Then, since Puz is an $\epsilon_{d,k,n}(t)$-strongly-difficulty interactive puzzle, we have that

$$\Pr\left(\mathsf{Exec}^{\text{DOS}}_{A,n,G_{m+1}}(k) = \textbf{true}\right) \leq \epsilon_{d,k,n}(t') \ . \tag{7}$$

Combining equations (4) through (7), we have that

$$\Pr\left(\mathsf{Exec}^{\text{DOS}}_{\mathcal{A},n,D(P)_{\mathsf{Puz},d,\mathsf{MAC},k}}(k) = \textbf{true}\right) \leq \epsilon_{d,k,n}(t') + |\mathsf{Servers}| \cdot \mathrm{Adv}^{\text{EUCMA}}_{\mathsf{MAC},k}(q_{\mathsf{Send}}, t') \ ,$$

where q_{Send} is the number of Send queries issued and $t' = t + t_0 q_{\mathsf{Send}}$, where t is the running time of $\mathcal{A}$ and t_0 is some constant specified by the protocol. Assuming $\mathrm{Adv}^{\text{EUCMA}}_{\mathsf{MAC},k}(q_{\mathsf{Send}}, t') \in \mathrm{negl}(k)$, we have

$$\Pr\left(\mathsf{Exec}^{\text{DOS}}_{\mathcal{A},n,D(P)_{\mathsf{Puz},d,\mathsf{MAC},k}}(k) = \textbf{true}\right) \leq \epsilon_{d,k,n}(t') + \mathrm{negl}(k) \ .$$

This shows part 1 of Defintion 3. Part 2 follows by inspection on Figure 1, under the assumption that none of the steps on lines 3–7 and 11–14 are expensive. Thus, $D(P)_{\mathsf{Puz},d,\mathsf{MAC},k}$ is an $\epsilon_{d,k,n}(t')$-DoS-resistant protocol. □