Security Notions and Generic Constructions for Client Puzzles

L. Chen[1], P. Morrissey[2], N.P. Smart[2] and B. Warinschi[2]

[1] Hewlett-Packard Laboratories,
Filton Road,
Stoke Gifford,
Bristol, BS34 8QZ,
United Kingdom.
liqun.chen@hp.com
[2] Computer Science Department,
Woodland Road,
University of Bristol,
Bristol, BS8 1UB,
United Kingdom.
{paulm, nigel, bogdan}@cs.bris.ac.uk

Abstract. Computational puzzles are mildly difficult computational problems that require resources (processor cycles, memory, or both) to solve. Puzzles have found a variety of uses in security. In this paper we are concerned with *client puzzles*: a type of puzzle used as a defense against Denial of Service (DoS) attacks. Before engaging in a resource consuming protocol with a client, a server demands that the client solves a freshly generated client puzzle. Despite their widespread use, the lack of formal models for security of client puzzles prevents a full analysis of proposed puzzles and, more importantly, prevents rigorous proofs for the effectiveness of puzzles as a DoS defense. The main contribution of this paper is a formal model for the security of client puzzles as a stepping stone towards solving the above problems. We clarify the interface that client puzzles should offer and give two security notions for puzzles. Both functionality and security are inspired by, and tailored to, the use of puzzles as a defense against DoS attacks. The first notion – puzzle unforgeability – requires that an adversary is unable to produce valid looking puzzles on its own. The second notion – puzzle-difficulty – requires that an adversary spends at least an appropriate amount of resources solving puzzles. Our definitions fill an important gap: breaking either of the two properties immediately leads to successful DoS attacks. We illustrate this point with an attack against a previously proposed puzzle construction. We show that a subtle flaw renders the construction forgeable and we explain how to exploit this flaw to mount a DoS attack on certain protocols that use this puzzle.
We also provide a generic construction of a client puzzle. Our construction uses a pseudorandom function family to provide unforgeability and a one way function for the difficulty. We prove our generic construction meets our definitions of unforgeability and difficulty for client puzzles. Finally, we discuss and analyze (in the random oracle model) a practical instantiation of our construction based on hash functions.

1 Introduction

A Denial of Service (DoS) attack on a server aims to render it unable to provide some service by depleting its internal resources. For example, the famous TCP-SYN flooding attack [9] prevents further connections to a server by starting a large number of TCP sessions which are then left uncompleted. The effort of the attacker is rather small, whereas the server quickly runs out of resources (which are allocated to the unfinished sessions).

One countermeasure against connection depletion DoS attacks uses client puzzles [14]. When contacted by some unauthenticated, potentially malicious client to execute some protocol and before allocating any resources for the execution, the server issues a client puzzle – a moderately hard computational problem. The server only engages in the execution of the protocol (and thus allocates resources) when the client returns a valid solution to the puzzle. The idea is that the server spends its resources only after the client has spent a significant amount of resources itself. To avoid the burden of running the above mechanism when no attackers are present, the defense only kicks in whenever the server resources drop below a certain threshold.

Client puzzles have received a lot of attention in the cryptographic community [2, 5, 10, 11, 14, 23, 26, 27] but most of the prior work consists of proposing puzzle constructions and arguing that those constructions do indeed work.

Although sometimes technical, such security arguments are with respect to intuitive security notions for puzzles since rigorous formal models for the security of such puzzles are missing. The absence of such models has (at least) two undesirable consequences. On the one hand the investigation of puzzle constructions usually concentrates on some security aspects and omits others which are of equal importance when puzzles are used as part of other protocols. More importantly, the absence of formal models prevents a rigorous, reduction-based analysis of the effectiveness of puzzles against DoS in the style of modern cryptography (where the existence of a successful DoS attacker implies the existence of an attacker against client puzzles).

In this paper we aim to solve the first problem outlined above as a first key step towards solving the second one. The main contribution of this paper is a formal framework for the design and analysis of client puzzles. In addition to fixing their formal syntax, we design security notions inspired by, and therefore tailored for, the use of client puzzles as a defense against DoS attacks. Specifically, we require that an adversary cannot produce valid puzzles on his own (*puzzle-unforgeability*) and that puzzles are non-trivial – the client needs indeed to spend at least a specified amount of resources to solve them – (*puzzle difficulty*). The use of client puzzles that do not fulfill at least one of our notions immediately leads to a successful DoS attack. Our definitions use well-established intuition and techniques for defining one-wayness and authentication properties. Apart from some design decisions regarding the measure for resources and the precise oracles an adversary should have access to, there are no deep surprises here. However, we highlight that the lack of rigorous definitions such as those we put forward in this paper is dangerous. Constructions that are secure at an intuitive level, may be in fact insecure when used. Indeed, we explicitly demonstrate that a popular construction, that does not meet our notion of unforgeability, does not protect and in fact facilitates DOS attacks in systems that use it.

Furthermore, we give a generic construction of a client puzzle that is secure in the sense we define. Many existing client puzzle constructions can be obtained as an instantiation of our generic construction, with only minor modifications if any. Our construction uses a pseudorandom function family to provide puzzle-unforgeability and puzzle-difficulty is obtained from a one-way function given a large part of the preimage. Difficulty here does not mean hardness of computation, but rather non-triviality. We prove our construction secure via an asymptotic reduction for unforgeability and a concrete reduction for difficulty. Next, we discuss our results in more details.

Our Contribution

FORMAL SYNTAX OF A CLIENT PUZZLE. Our first contribution is a formal syntax for client puzzles. We define a client puzzle as a tuple of algorithms for system setup, puzzle generation, solution finding, puzzle authenticity checking, and solution checking. The definition is designed to capture the main functionality required from client puzzles from the perspective of their use against DoS attacks.

SECURITY NOTIONS FOR CLIENT PUZZLES. The use of puzzles against DoS attacks also inspired the two (orthogonal) security notions for client puzzles that we design.

To avoid storing puzzles handed out to clients (a resource consuming task), the server gives puzzles away and expects the client to hand back both the puzzle and its solution. Obviously, the server needs to be sure the client cannot produce puzzles on its own, as this would lead to trivial attacks. We remark that this aspect is often overlooked in existing constructions since it is only apparent when puzzles are considered in the precise context for which they are intended. We capture this requirement via the notion of puzzle-unforgeability. Formally, we define a security game where the adversary is given certain querying capabilities (he can for example request to see puzzles and their solutions, can verify the authenticity of puzzles, etc) and aims to output a new puzzle which the server deems as valid.

The second notion, puzzle-difficulty, reflects the idea that the client needs to spend a certain amount of resources to solve a puzzle. In our definition we took adversary resources to mean "clock cycles", as this design decision allows us to abstract away undesirable details like the distributed nature of many DoS adversaries. We define a security game where the adversary is given various querying capabilities sufficient for mimicking a DoS attack-like environment: he can see puzzles and their solutions, obtain solutions for puzzles he chooses, etc. The challenge for the adversary is to solve a given challenge puzzle spending less than a certain number of clock cycles, with probability better than a certain threshold.

AN ATTACK ON THE JUELS AND BRAINARD PUZZLES. Most of the previous work on puzzles concentrates exclusively on the difficulty aspect and overlooks, or only partially considers, the unforgeability property. One such work is the puzzle construction proposed by Juels and Brainard [14]. We demonstrate the usefulness of our definitions by showing the Juels and Brainard construction is forgeable. We then explain how a system using this kind of puzzle can be attacked by exploiting the weakness we have identified.

GENERIC CONSTRUCTIONS. We provide a generic construction of a client puzzle inspired by the Juels and Brainard sub-puzzle construction [14]. First, we evaluate a pseudorandom function (PRF), keyed by some secret value, on inputs including a random nonce, hardness parameter and a system specific string. This stage ensures uniqueness of the puzzle and the desired unforgeability; only the server that possesses the hidden key is able to perform this operation and hence generate valid puzzles. The remaining information to complete the puzzle is then computed by evaluating a one way function (OWF), for which finding preimages has a given difficulty, on the output of the PRF; the goal in solving the puzzle is to find such a preimage given the inputs to the PRF and the target. The idea is that the client would need to do an exhaustive search on the possible preimage space to find such a preimage. We certify the intuition by rigorous proofs that the generic construction meets the security definitions that we put forth, for appropriately chosen parameters. Importantly, many *secure variants* of previously proposed constructions can be obtained as instances of our generic construction. For example, the puzzle constructions proposed by Juels and Brainard [14] puzzle and the two-party variant of the Waters et al. puzzles [27] can be seen as variants of our generic construction. Finally, we provide concrete security bounds for the first of these puzzles. We do so in the random oracle model which we use to obtain secure and efficient instantiations of the two primitives used by our generic construction.

Related Work

MERKLE PUZZLES. The use of puzzles in cryptography was pioneered by Merkle [18] who used puzzles to establish a secret key between parties over an insecure channel. Since then the optimality of Merkle puzzles has been analyzed by Impagliazzo and Rudich [12] and Barak and Mahmoody–Ghidary [3]. The possibility of basing weak public key cryptography on one-way functions, or some variant of them was recently explored by Biham, Goren and Ishai [4]. Specifically, a variant of Merkle's protocol is suggested whose security is based on the one-wayness of the underlying primitive.

CLIENT PUZZLES. Client puzzles were first introduced as a defense mechanism against DoS attacks by Juels and Brainard [14]. The construction they proposed uses hash function inversion as the source of puzzle-difficulty. They also attempt to obtain puzzle-unforgeability but partially fail in two respects. By neglecting the details of how puzzles are to be used against a DoS attack, the construction suffers from a flaw (which we explain how to exploit later in this paper) that can be used to mount a DoS attack. Secondly, despite intuitive claims that security is based on the one-wayness of the hash function used in the construction, security requires much stronger assumptions, namely one-wayness with partial information about the preimage. The authors also present a method to combine a key agreement or authentication protocol with a client puzzle, and present a set of informal desirable properties of puzzles. Building on this work, Aura et al. [2] use the same client puzzle protocol construction but present a new client puzzle mechanism, also based on hash function inversion, and extend the set of desirable properties.

An alternative method for constructing client puzzles and client puzzle protocols was proposed by Waters et al. [27]. This technique assumes the client puzzle protocol is a three party protocol and constructs a client puzzle based on the discrete logarithm problem for which authenticity and correctness can be verified using a Diffie–Hellman based technique. One of the main advantages of this approach is that puzzle generation can be outsourced from the server to another external bastion, yet verification of solutions can be performed by the server itself.

More recently Tritilanunt et al.[26] proposed a client puzzle based on the subset sum problem. Schaller et al. [23] have also used what they refer to as cryptographic puzzles for broadcast authentication in networks.

An interesting line of work analyzes ways to construct stronger puzzles out of weaker ones. The concept of chaining together client puzzles to produce a new and more difficult client puzzle was introduced by Groza and Petrica [11]. Their construction enforces a sequential solving strategy, and thus yields a harder puzzle. A related work is that of Canetti, Halevi, and Steiner [5] who are concerned with relating the difficulty of solving one single puzzle to that of solving several independent puzzles. They consider the case of "weakly"verifiable puzzles (puzzles for which the solution can only be checked by the party that produced the puzzles). That paper does not consider the use of puzzles in the context of DoS attacks, and thus is not concerned with authenticity.

DoS ATTACKS. A classification of remote DoS attacks, countermeasures and a brief consideration of Distributed Denial of Service (DDoS) attacks were given by Karig and Lee [15]. Following this Specht and Lee [25] give a classification of DDoS attacks, tools and countermeasures. In [25] the adversarial model of [15] is extended to include Internet Relay Chat (IRC) based models. The authors of [25] also classify the types of software used for such attacks and the most common known countermeasures. Other classifications of DDoS attacks and countermeasures were later given by [7, 19].

A number of protocols have been designed to resist DDoS attacks. The most important examples are the JFK protocol [1] and the HIP protocol [20]. The JFK protocol of Aiello et al. [1] trades the forward secrecy property, known as adaptive forward secrecy, for denial of service resistance. The original protocol does not use client puzzles. In [24] the cost based technique of Meadows [16, 17] is used to analyze the JFK protocol. Two denial of service attacks are found and both can be prevented by introducing a client puzzle into the JFK protocol.

SPAM AND TIME-LOCK CRYPTO. Other proposals for the use of puzzles include the work of Dwork and Naor who propose to use pricing function (a particular type of puzzles) to combat junk email [8]. The basic principle is the pricing function costs a given amount of computation to compute and this computation can be verified cheaply without any additional information. A service provider could then issue a "stamp duty" on bulk mailings. Finally, Rivest et al. introduced the notion of timed-release crypto in [21] and instantiate this notion with a time-lock puzzle. The overall goal of timed-release crypto is to encrypt a message such that nobody, even the sender, can decrypt it before a given length of time has passed.

Paper Overview We start with a sample client puzzle from Section 2. Our formal definition of a client puzzle and a client puzzle protocol is in Section 3. In Section 4 we give security notions for client puzzles in terms of unforgeability and difficulty. We demonstrate that the Juels and Brainard client puzzles is insecure in Section 5. Finally, our generic construction of a client puzzle is given in Section 6. We also include a sample instantiation based on hash functions which we analyze in the random oracle model. The Appendix contains additional notation and deferred proofs.

2 An Example Client Puzzle

As an example of a client puzzle we give a brief description of the puzzle generation process for the Juels and Brainard construction [14]. In our description we refer to the (authorized) puzzle generation entity (or user) as the *generator* and the (authorized) puzzle solving entity (or user) as the *solver*. We use the term "puzzle" from here onwards for individual puzzle instances. We write $\{0,1\}^t$ for the set of binary strings of length t and $\{0,1\}^*$ for the set of binary strings of arbitrary finite length. If $x = x_0, x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n$ is a bit string then we let $x\langle i,\ j\rangle$ denote the sub string $x_i, \ldots, x_j$.

For this construction the generator (generally some server) holds a long term secret value s chosen uniformly from a space large enough to prevent exhaustive key search attacks. The server also chooses a hardness parameter: a pair $Q = (\alpha, \beta) \in \mathbb{N}^2$ which ensure puzzles have a certain amount of difficulty to solve. We let $H : \{0,1\}^* \mapsto \{0,1\}^m$ be some hash function. To generate a new puzzle the generator performs the following steps to compute the required sub-puzzle instances P_j for $j \in \{1, 2, \ldots, \beta\}$:

- A bit string σ_j is computed as $\sigma_j = H(s, \mathsf{str}, j)$. The value str has the structure $\mathsf{str} = t \| M$ for t some server time value[1] and M some unique value[2]. We denote $x_j = \sigma_j\langle 1,\ \alpha\rangle$ and $z_j = \sigma_j\langle \alpha+1,\ m\rangle$.
- A value y_j is computed as $y_j = H(\sigma_j)$ and the sub-puzzle instance is $P_j = (z_j, y_j)$.

The full puzzle instance is then the required parameters plus the tuple of sub-puzzle instances $\mathsf{puz} = (Q, \mathsf{str}, P = (P_1, P_2, \ldots, P_\beta))$. The sub-puzzle instance generation process is summarized in Figure 1.

A solution to a given sub-puzzle P_j is any string x'_j such that $H(x'_j \mid\mid z_j) = y_j$. The solution to the full puzzle instance is a tuple of solutions to the sub-puzzles. To verify a potential solution $\mathsf{soln} = (Q, \mathsf{str}, \mathsf{soln}_1, \cdots, \mathsf{soln}_\beta)$ the generator verifies each P_j and soln_j by checking that $H(\mathsf{soln}_j \mid\mid z_j) = y_j$ for each j. The authenticity of a given puzzle is checked by regenerating each P_j using s and comparing this to the puzzle submitted.

[1] The details of the type of value this is are not described in [14] but here we will assume this is as a bit string.

[2] In [14] this is specified as the first message flow of a protocol or some other unique data. Again, we will assume this is encoded as a bit string since this is not specified.

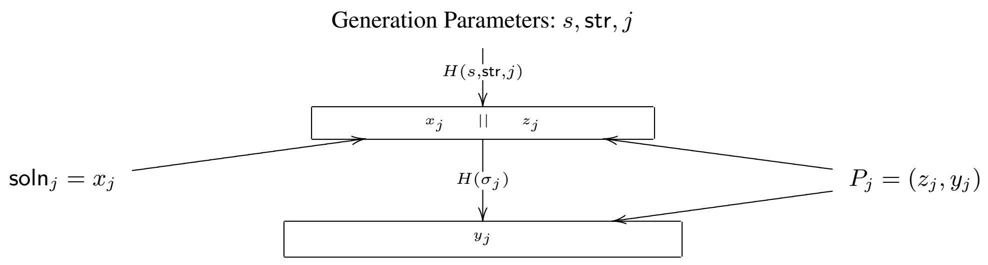


Fig. 1. The Juels and Brainard Sub-Puzzle Instance Generation.

To incorporate this client puzzle into a client puzzle protocol the server allocates buffer slots, by using a hash table on the values of M, for each puzzle and correct solution submitted. This ensures that only one puzzle instance and solution are accepted for a given value of M.

3 Client Puzzles

The role of a client puzzle in a protocol is to give one party some assurance that the other party has spent at least a given amount of effort computing a solution to a given puzzle instance. In this section we give a formal definition of a client puzzle in the most general sense.

FORMAL SYNTAX OF A CLIENT PUZZLE. A client puzzle is a tuple of algorithms: a setup algorithm for generating long term public and private parameters, an algorithm for generating puzzle instances of a given difficulty, a solution finding algorithm, an algorithm for verifying authenticity of a puzzle instance and an algorithm for verifying correctness of puzzle and solution pairs. We formally define a client puzzle as follows.

Definition 1 (Client Puzzle). *A* client puzzle $\mathsf{CPuz} = (\mathsf{Setup}, \mathsf{GenPuz}, \mathsf{FindSoln}, \mathsf{VerAuth}, \mathsf{VerSoln})$ *is given by the following algorithms:*

- Setup *is a* p.p.t. *setup algorithm. On input* 1^k*, for security parameter* k*, it performs the following operations:*
  - *Selects the long term secret key space* sSpace*, hardness space* QSpace*, string space* $\mathsf{strSpace}$*, puzzle instance space* $\mathsf{puzSpace}$ *and solution space* $\mathsf{solnSpace}$.
  - *Selects the long term puzzle generation key* $s \xleftarrow{\$} \mathsf{sSpace}$.
  - *Assigns* $\mathsf{params} \leftarrow (\mathsf{sSpace}, \mathsf{puzSpace}, \mathsf{solnSpace}, \mathsf{QSpace}, \Pi)$ *and outputs* (params, s).

  The tuple params *is the public system parameters and as such is not explicitly given as an input to other algorithms. The value* s *is kept private by the puzzle generator and* $\Pi \in \mathsf{params}$ *is any additional public information, such as descriptions of algorithms, required for the client puzzle.*
- GenPuz *is a* p.p.t. *puzzle generation algorithm. On input* $s \in \mathsf{sSpace}$*,* $Q \in \mathsf{QSpace}$ *and* $\mathsf{str} \in \mathsf{strSpace}$ *it outputs a puzzle instance* $\mathsf{puz} \in \mathsf{puzSpace}$.
- $\mathsf{FindSoln}$ *is a probabilistic solution finding algorithm. On inputs* $\mathsf{puz} \in \mathsf{puzSpace}$ *and a run time* $\tau \in \mathbb{N}$ *it outputs a* potential solution $\mathsf{soln} \in \mathsf{solnSpace}$ *after at most* τ *clock cycles of execution.*
- $\mathsf{VerAuth}$ *is a* d.p.t. *puzzle authenticity verification algorithm. On inputs* $s \in \mathsf{sSpace}$ *and* $\mathsf{puz} \in \mathsf{puzSpace}$ *this outputs* $\mathbf{true}$ *or* $\mathbf{false}$.
- $\mathsf{VerSoln}$ *is a deterministic solution verification algorithm. On inputs* $\mathsf{puz} \in \mathsf{puzSpace}$ *and a potential solution* $\mathsf{soln} \in \mathsf{solnSpace}$ *this outputs* $\mathbf{true}$ *or* $\mathbf{false}$.

For correctness we require that if $(\mathsf{params}, s) \leftarrow \mathsf{Setup}(1^k)$ *and* $\mathsf{puz} \leftarrow \mathsf{GenPuz}(s, Q, \mathsf{str})$*, for* $Q \in \mathsf{QSpace}$ *and* $\mathsf{str} \in \mathsf{strSpace}$*, then*

- $\mathsf{VerAuth}(s, \mathsf{puz}) = \mathbf{true}$ *and*
- $\exists\, \tau \in \mathbb{N}$ *such that* $\mathsf{soln} \leftarrow \mathsf{FindSoln}(\mathsf{puz}, \tau)$ *and* $\mathsf{VerSoln}(\mathsf{puz}, \mathsf{soln}) = \mathbf{true}$.

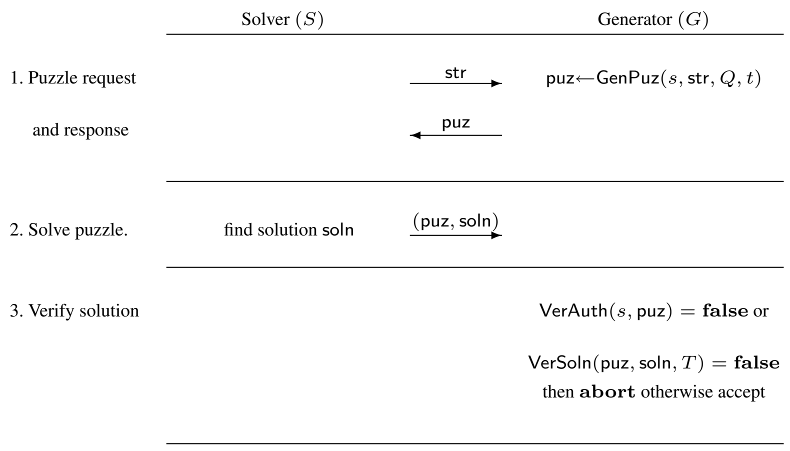
Remark 1. Typically client puzzles use a set of system parameters, most notably system time, as input to the puzzle generation algorithm. This is so the server has a mechanism for expiring puzzles handed out to clients. In our model we use str to capture this input and do not enforce any particular structure on it.

Remark 2. To prevent DoS attacks that exhaust the server memory it is desirable that the server stores as little state as possible for uncompleted protocol runs (i.e. before a puzzle has been solved). We refer to this concern of client puzzles as "state storage costs" [2]. We build this into our definition of a client puzzle by insisting that *only* a single value of s is stored by a server; all the data necessary to solve a given puzzle and to re-generate, and hence verify authenticity of a puzzle and solution pair, is included in the puzzle description puz.

Remark 3. Generally, for a puzzle to be "secure" when used within a client puzzle protocol, we want puzzles generated to be unique and for puzzle and solution pairs to only be validly used once by a client. In actual usage, a server can filter out resubmitted correctly solved puzzle and solution pairs by, for example, using a hash table mechanism. Uniqueness of puzzles can be ensured by having GenPuz select a random nonce n_S and use this in the puzzle generation.

Remark 4. Our definition assumes private verifiability for $\mathsf{VerAuth}$. Generally the only party concerned with checking who generated a given puzzle is the puzzle generator (client puzzles are used *before* any other transactions take place and to protect the generator and no other party). Although in some cases it may be useful to have publicly verifiable puzzles it would complicate the definition and we choose to keep our definition practical yet as simple as possible.

CLIENT PUZZLE PROTOCOLS. Intuitively, a client puzzle protocol is a protocol which uses a client puzzle in its message flows to provide DoS resistance. We first allow a one time setup during which the generator G runs $\mathsf{Setup}(1^k)$ for security parameter k to obtain params and s. We then define $\Pi(\mathsf{CPuz})$, the client protocol between a solver S and G, to be the message flows of Figure 2.



Solver (S)
Generator (G)
1. Puzzle request and response
str
$\mathsf{puz} \leftarrow \mathsf{GenPuz}(s, \mathsf{str}, Q, t)$
puz
2. Solve puzzle.
find solution soln
$(\mathsf{puz}, \mathsf{soln})$
3. Verify solution
$\mathsf{VerAuth}(s, \mathsf{puz}) = \mathbf{false}$ or
$\mathsf{VerSoln}(\mathsf{puz}, \mathsf{soln}, T) = \mathbf{false}$
then $\mathbf{abort}$ otherwise accept

Fig. 2. The Client Puzzle Protocol $\Pi(\mathsf{CPuz})$ constructed from CPuz

One way to use a client puzzle protocols as a counter measure against connection depletion DoS attacks is to stack $\Pi(\mathsf{CPuz})$ on top of the original protocol in some manner. We do not consider the specifics of how this can be achieved here. In order that the use of client puzzles does not introduce malformed packet DoS attacks [16, 17] it is important that each of the algorithms in a given client puzzle is as efficient as possible. Ideally it should take more time for an adversary to spoof a given message flow in Figure 2 than it would to check correctness or authenticity of such spoofed messages.

4 Security Notions for Client Puzzles

We define two notions for client puzzles. The first measures the ability of an adversary to produce a correctly authenticating puzzle with an unknown private key. We refer to this as the ability of an adversary to forge a client puzzle. The second notion gives a measure of the likelihood of an adversary finding a solution to a given puzzle within a given number of clock cycles of execution. We refer to this as the difficulty of a client puzzle. Intuitively, these are both what one would expect to require from a client puzzle given its role in defenses against DoS attacks; being able to either forge puzzles or solve them faster than expected allows an adversary to mount a DoS attack.

We first review the definition of a function family since we will use function families to express security of a given client puzzle in terms of difficulty. A *function family* is a map $F : I \times D \mapsto R$. The set I is the set of all possible indices, D the domain and R the range. Unless otherwise specified we assume $I = \mathbb{N}$. The set R is finite and all sets are nonempty. We write $F_i : D \mapsto R$ for $F_i(d) = F(i, d)$ where $i \in I$ and refer to F_i as an instance of F.

UNFORGEABILITY OF PUZZLES. We first define our notion of unforgeability of client puzzles. Intuitively, we require an adversary that sees puzzles generated by the server (possibly together with their associated solutions), and that can verify the authenticity of any puzzle it chooses, cannot produce a valid looking puzzle on his own.

To formalize unforgeability of a client puzzle we use the following game $\mathsf{Exec}^{\mathsf{UF}}_{\mathcal{A},\mathsf{CPuz}}(k)$ between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.

(1) The challenger $\mathcal{C}$ first runs Setup on input 1^k to obtain (params, s). The tuple params is given to $\mathcal{A}$ and s is kept secret by $\mathcal{C}$.

(2) The adversary $\mathcal{A}$ gets to make as many $\mathsf{CreatePuz}(Q, \mathsf{str})$ and $\mathsf{CheckPuz}(\mathsf{puz})$ queries as it likes which $\mathcal{C}$ answers as follows.

- $\mathsf{CreatePuz}(Q, \mathsf{str})$ queries. A new puzzle is generated $\mathsf{puz} \leftarrow \mathsf{GenPuz}(s, Q, \mathsf{str})$ and output to $\mathcal{A}$.
- $\mathsf{CheckPuz}(\mathsf{puz})$ queries. If $\mathsf{VerAuth}(s, \mathsf{puz}) = \mathbf{true}$ and puz was not output by $\mathcal{C}$ in response to a $\mathsf{CreatePuz}$ query then $\mathcal{C}$ terminates the game setting the output to 1. Otherwise $\mathbf{false}$ is returned to $\mathcal{A}$.

(3) If $\mathcal{C}$ does not terminate the game in response to a Check query then eventually $\mathcal{A}$ terminates and the output of the game is set to 0.

We say the adversary $\mathcal{A}$ wins if $\mathsf{Exec}^{\mathsf{UF}}_{\mathcal{A},\mathsf{CPuz}}(k) = 1$ and loses otherwise. We define the advantage of such an adversary as

$$\mathbf{Adv}^{\mathsf{UF}}_{\mathcal{A},\mathsf{CPuz}}(k) = \Pr\Big[\mathsf{Exec}^{\mathsf{UF}}_{\mathcal{A},\mathsf{CPuz}}(k) = 1\Big].$$

Puzzle-unforgeability then means that no efficient adversary can win the above game with non-negligible probability.

Definition 2 (Puzzle-unforgeability). *A client puzzle* CPuz *is* UF *secure if for any* p.p.t. *adversary* $\mathcal{A}$ *its advantage* $\mathbf{Adv}^{\mathsf{UF}}_{\mathcal{A},\mathsf{CPuz}}(k)$ *is a negligible function of k.*

Remark 1. In the game $\mathsf{Exec}^{\mathsf{UF}}_{\mathcal{A},\mathsf{CPuz}}(k)$ we allow $\mathcal{A}$ access to all algorithms defined in CPuz. In particular, we allow unlimited access to the GenPuz algorithm for any given chosen inputs. This allows $\mathcal{A}$ to generate as many puzzles as it wishes (since $\mathcal{A}$ is p.p.t. it will anyway generate at most polynomially many) with any given chosen key and difficulty values. Notice that the adversary can find solutions to any puzzle by running the $\mathsf{FindSoln}$ algorithm which is public. These abilities are sufficient to mimic the environment in which a DoS attacker would sit.

DIFFICULTY OF SOLVING PUZZLES. We formalize the idea that a puzzle CPuz cannot be solved trivially via the game $\mathsf{Exec}^{Q,\mathsf{DIFF}}_{\mathcal{A},\mathsf{CPuz}}(k)$ between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. The game is defined for each hardness parameter $Q \in \mathbb{N}$ as follows:

(1) The challenger $\mathcal{C}$ runs Setup on input 1^k to obtain (params, s) and passes params to $\mathcal{A}$.

(2) The adversary $\mathcal{A}$ is allowed to make any number of $\mathsf{CreatePuzSoln}(\mathsf{str})$ queries throughout the game. In response to each such query $\mathcal{C}$ generates a new puzzle as $\mathsf{puz} \leftarrow \mathsf{GenPuz}(s, Q, \mathsf{str})$ and finds a solution soln such that $\mathsf{VerSoln}(\mathsf{puz}, \mathsf{soln}) = \mathbf{true}$. The pair $(\mathsf{puz}, \mathsf{soln})$ is then output to $\mathcal{A}$.

(3) At any point during the execution $\mathcal{A}$ is allowed to make a single $\mathsf{Test}(\mathsf{str}^\dagger)$ query. In response, the challenger generates a challenge puzzle as $\mathsf{puz}^\dagger \leftarrow \mathsf{GenPuz}(s, Q, \mathsf{str}^\dagger)$ which it returns to $\mathcal{A}$.

Adversary $\mathcal{A}$ terminates its execution by outputting a potential solution $\mathsf{soln}^\dagger$. We define the running time τ of $\mathcal{A}$ as being the running time of all of the experiment $\mathsf{Exec}^{Q,\mathsf{DIFF}}_{\mathcal{A},\mathsf{CPuz}}(k)$.

We say the adversary wins $\mathsf{Exec}^{Q,\mathsf{DIFF}}_{\mathcal{A},\mathsf{CPuz}}(k)$ if $\mathsf{VerSoln}(\mathsf{puz}^\dagger, \mathsf{soln}^\dagger) = \mathbf{true}$. In this case we set the output of $\mathsf{Exec}^{Q,\mathsf{DIFF}}_{\mathcal{A},\mathsf{CPuz}}(k)$ to be 1 and otherwise to 0. We then define the success of an adversary $\mathcal{A}$ against CPuz as

$$\mathbf{Succ}^{Q,\mathsf{DIFF}}_{\mathcal{A},\mathsf{CPuz}}(k) = \Pr\Big[\mathsf{Exec}^{Q,\mathsf{DIFF}}_{\mathcal{A},\mathsf{CPuz}}(k) = 1\Big].$$

We define the difficulty of puzzle solving by requiring that for any puzzle hardness the success of any adversary that runs in a bounded number of steps falls bellow a certain threshold (that is related to the hardness of the puzzle).

Definition 3 (Puzzle-difficulty). *Let $\varepsilon : \mathbb{N}^2 \mapsto \big(\mathbb{N} \mapsto [0,1]\big)$ be a family of monotonically increasing functions. We use the notation $\varepsilon_{k,Q}(\cdot)$ for the function within this family corresponding to security parameter k and hardness parameter Q. We say a client puzzle CPuz is $\varepsilon(\cdot)$–DIFF if for all $\tau \in \mathbb{N}$, for all adversaries $\mathcal{A}$ in $\mathsf{Exec}^{\mathsf{DIFF},Q}_{\mathcal{A},\mathsf{CPuz}}(k)$, for all security parameters $k \in \mathbb{N}$, and for all $Q \in \mathbb{N}$ it holds that*

$$\mathbf{Succ}^{Q,\mathsf{DIFF}}_{\mathcal{A}_\tau,\mathsf{CPuz}}(k) \leq \varepsilon_{k,Q}(\tau)$$

where $\mathcal{A}_\tau$ is the adversary $\mathcal{A}$ restricted to at most τ clock cycles of execution.

Remark 1. The security game above allows $\mathcal{A}$ to obtain many puzzle and solution pairs by making queries to model actual usage in DoS settings; when a client puzzle is used as part of a client puzzle protocol an adversary may see many such puzzles and solutions exchanged between a given generator and solver on a network. The adversary could then learn something from these.

Remark 2. In the definition of the Test query, we do allow the string $\mathsf{str}^\dagger$ to be one previously submitted as a $\mathsf{CreatePuzSoln}$ query and allow $\mathsf{CreatePuzSoln}$ queries on any string including $\mathsf{str}^\dagger$ after the Test query. It then immediately follows that a difficult puzzle needs to be such that each puzzle generated is unique. Otherwise, a previously obtained solution through the $\mathsf{CreatePuzSoln}$ query may serve as a solution to the challenge query. Furthermore, it also follows that solutions to some puzzles should not be related to the solutions of other puzzles, as otherwise a generalization of the above attack would work.

Remark 3. The queries $\mathsf{CreatePuz}$ (used in the game for puzzle-unforgeability) and $\mathsf{CreatePuzSoln}$ used in the above game are related, but different. The query $\mathsf{CreatePuzSoln}$ outputs a puzzle together with its solution. The second is more subtle: in a $\mathsf{CreatePuz}$ query we allow $\mathcal{A}$ to specify the value of Q used but in $\mathsf{CreatePuzSoln}$ we do not (the value of Q is fixed throughout the difficulty game).

Remark 4. Clearly any puzzle that is $\varepsilon(\cdot)$–DIFF is also $(\varepsilon(\cdot) + \mu)$-$\mathsf{DIFF}$ where $\mu \in \mathbb{R}_{>0}$ is such that $\varepsilon(\tau) + \mu \leq 1$ (since $\mathbf{Succ}^{Q,\mathsf{DIFF}}_{\mathcal{A}_\tau,\mathsf{CPuz}}(k) \leq \varepsilon_{k,Q}(\tau) \leq \varepsilon_{k,Q}(\tau) + \mu$). The most accurate measure of difficulty for a given puzzle CPuz is then the function $\varepsilon(\tau) = \inf_{\mathcal{A}_\tau} \mathbf{Succ}^{Q,\mathsf{DIFF}}_{\mathcal{A}_\tau,\mathsf{CPuz}}(k)$.

Remark 5. Since we measure the running time of the adversary in clock cycles, the model abstracts away the possibility that the adversary may be distributed and thus facilitates further analysis (for example of the effectiveness of client puzzle defense against DoS attacks).

5 An Attack on the Juels and Brainard Puzzles

In this section we describe an attack on the Juels and Brainard [14] client puzzle mechanism as described in Section 2. The attack works because puzzles are forgeable, which is due to a crucial weakness in puzzle generation; each set of generation parameters defines a family of puzzles each with a different hardness value. Finally we construct a DDoS attack on servers using certain client puzzle protocols based on this construction. This attack clearly demonstrates the applicability of our definitions and how they can be used to find problems with a given client puzzle construction.

PROVING FORGEABILITY. The reason the construction is forgeable is the authentication is not unique to a given instance but covers a number of instances of varying difficulty. This occurs because the puzzle instance difficulty is not included in the first preimage of the sub-puzzle construction. We exploit this weakness and construct an adversary $\mathcal{A}$ with $\mathbf{Adv}^{\mathsf{UF}}_{\mathcal{A},\mathsf{CPuz}}(k) = 1$.

We have the following lemma regarding the forgeability of the Juels and Brainard construction.

Lemma 1. *The client puzzle construction of Juels and Brainard [14] is* not UF *secure.*

Proof. To prove this we construct an adversary $\mathcal{A}$ against the UF security of the construction that can win the security game $\mathsf{Exec}^{\mathsf{UF}}_{\mathcal{A},\mathsf{CPuz}}(k)$ with probability 1. We now describe the details of $\mathcal{A}$.

At the start of the security game $\mathcal{A}$ is given a set of public parameters. The adversary then makes a query $\mathsf{CreatePuz}(Q, \mathsf{str})$ for some random choices of Q and str where $Q = (\alpha, \beta)$ and receives a puzzle instance $\mathsf{puz} = (Q, \mathsf{str}, P = (P_1, P_2, \ldots, P_\beta))$ in response. Next $\mathcal{A}$ removes the first bit of each P_i to obtain $P_i^\dagger$ and constructs $Q^\dagger = (\alpha + 1, \beta)$ and $\mathsf{puz}^\dagger = (Q^\dagger, \mathsf{str}, P^\dagger = (P_1^\dagger, P_2^\dagger, \ldots, P_\beta^\dagger))$. The adversary then makes a query $\mathsf{CheckPuz}(\mathsf{puz}^\dagger)$.

Clearly $\mathcal{A}$ wins with probability 1 since puz and $\mathsf{puz}^\dagger$ are both generated from the same s and str hence $\mathsf{puz}^\dagger$ will correctly verify yet was not output from a $\mathsf{CreatePuz}$ query. □

Remark 1. One could also prove Lemma 1 by having the adversary construct the forgery as $Q^\dagger = (\alpha, \beta - 1)$ and then $\mathsf{puz}^\dagger = (Q^\dagger, \mathsf{str}, (P_1^\dagger, \ldots, P_{\beta-1}^\dagger))$. One could also vary the number of bits moved between α and each P_i or change the number of sub-puzzles deleted. The reason we choose to give the proof in the manner given is because this specific method allows for the construction of a DDoS attack with the given assumptions we make about the protocol using this particular client puzzle. We describe this attack next.

CONSTRUCTING A DDOS ATTACK. We now use the forgeability of the construction to mount a DDoS attack on client puzzle protocols based on this client puzzles. The attack works when the difficulty parameter is increased in a certain way and when the hash table, mentioned in [14] and used to prevent multiple puzzle instance and solution submissions, is based on some unique data for each instance that is not in the preimage of any sub-puzzle. A hash table mechanism that depends on some unique data contained in each sub-puzzle preimage, as is mentioned in [14], would thwart the following DDoS attack on client puzzle protocols based on this client puzzle.

We first assume the client puzzle is used in the client puzzle protocol of [14] and the generator increases Q by increasing α many times for each increase in β. We also assume any hash tables used are computed using either the puzzle instance alone or the correct solutions alone.

To mount the DDoS attack the adversary commands each of its zombies (platforms the adversary controls) to start a run of the protocol with the server under attack. The server will begin to issue puzzle instances and then, when enough requests are received, will increase Q by incrementing α. Each zombie computes a solution to the first puzzle it receives and to submits this to the server. Then, while this puzzle has not expired, each time α is incremented, a new puzzle and solution pair is trivially computed by removing the first bit from each x_i and concatenating this to the end of each soln_i previously computed. The new puzzle and solution pair are submitted to the server and will correctly verify and will then be allocated buffer space (due to our assumptions on the hash table mechanism). When a zombies' puzzle expires it obtains a new one. As the value Q is increased then so will the puzzle expiry period and hence more forged puzzles can be used per valid puzzle obtained eventually exhausting the memory resources of the server.

Also, even if we assume that the buffer allocation based on the hash table mechanism is as in [14] the attack will still consume a huge amount of server computational resources. This is because the adversary can trivially spoof new puzzle instances and solutions from previous ones. These will not be allocated buffer space due to the hash table mechanism, but will consume computation via server verification computations. In the next section we give a an example instantiation of a generic construction that is a repaired version of the sub-puzzle mechanism; an unforgeable version of the sub-puzzle construction.

6 A Generic Client Puzzle Construction

In this section we provide a generic construction for a client puzzle which also repairs the flaw identified in the previous section with respect to the Juels and Brainard puzzle. Our construction is based on a pseudorandom function (PRF)

and a one way function (OWF). We prove our generic construction is secure according to the definitions we put forth in this paper, and show one possible instantiation. Intuitively, the unforgeability of puzzles is ensured by the use of the PRF and the difficulty of solving puzzles is ensured by the hardness of inverting the one-way function. We first review some notational conventions and definitions regarding function families, pseudorandom functions, and concrete notions for pseudorandom function families and one way function families.

If F is a function family then we use the notation $f \xleftarrow{\$} F$ for $i \xleftarrow{\$} I;\ f{\leftarrow}F_i$. We denote the set of all possible functions mapping elements of D to R by $\mathsf{Func}(D, R)$. A *random function* from D to R is then a function selected uniformly at random from $\mathsf{Func}(D, R)$.

PSEUDORANDOMNESS. We define the PRF (pseudorandom function) game $\mathsf{Exec}^{\mathsf{PRF},b}_{\mathcal{B},F}(k)$ for an adversary $\mathcal{B}$ against the function family $F : \mathcal{K} \times D \mapsto R$, where $|\mathcal{K}| = 2^k$, as follows.

(1) For $b = 1$ the adversary $\mathcal{B}$ has black box access to a truly random function $\mathcal{R}$ from the set $\mathsf{Func}(D, R)$ and for $b = 0$ the adversary $\mathcal{B}$ has black box access to a function F_s chosen at random from F.

(2) The adversary $\mathcal{B}$ is allowed to ask as many queries as it wants to whichever function it has black box access to. Eventually $\mathcal{B}$ terminates outputting a bit b^*.

We set the output of $\mathsf{Exec}^{\mathsf{PRF},b}_{\mathcal{B},F}(k)$ to 1 if $b^* = b$ and set the output to 0 otherwise. We then define the advantage of an adversary against F in terms of PRF as

$$\mathbf{Adv}^{\mathsf{PRF}}_{\mathcal{B},F}(k) = \left|\Pr\left[\mathsf{Exec}^{\mathsf{PRF},0}_{\mathcal{B},F}(k) = 1\right] - \Pr\left[\mathsf{Exec}^{\mathsf{PRF},1}_{\mathcal{B},F}(k) = 1\right]\right|.$$

CONCRETE PSEUDORANDOM AND ONE WAY FUNCTION FAMILIES. Here we briefly review concrete notions of security for pseudorandom function and one way function families. We depart from the typical "(ε, t)- hardness" style of definitions, as they are not sufficient for our purposes. Instead we view ε, the probability of a break, as a function of the running time τ of the adversary. So, a primitive is $\varepsilon(\cdot)$- secure if for all adversaries running in time τ the probability of breaking the primitive is at most $\varepsilon(\tau)$.

Definition 4 **($\nu_k(\cdot)$–PRFF).** *Let $F : \mathcal{K} \times D \mapsto R$ be a function family and $\nu : \mathbb{N} \mapsto \big(\mathbb{N} \mapsto [0, 1]\big)$ be a family of monotonically increasing functions. We say F is a $\nu_k(\cdot)$–PRFF if for all $k \in \mathcal{K}$ and for all adversaries $\mathcal{A}$ it holds that $\mathbf{Adv}^{\mathsf{PRF}}_{\mathcal{A}_\tau,F}(k) \le \nu_k(\tau)$.*

Note that, in the definition of an $\nu_k(\cdot)$–PRFF, the security parameter k specifies the size of the keyspace for the game $\mathsf{Exec}^{\mathsf{PRF},b}_{\mathcal{A}_\tau,F}(k)$ and the actual key, and hence function from the family used, is chosen at random from this keyspace.

Definition 5 **($\varepsilon_i(\cdot)$–OWF).** *For an adversary $\mathcal{A}$ we define its advantage against a function $\psi : \mathcal{X} \mapsto \mathcal{Y}$, where $\mathcal{X}$ is fixed and finite, in terms of OWF as*

$$\mathbf{Adv}^{\mathsf{OWF}}_{\mathcal{A},\psi} = \Pr[x \xleftarrow{\$} \mathcal{X};\ y{\leftarrow}\psi(x);\ \ (\tilde{x}{\leftarrow}\mathcal{A}(y) \wedge \psi(\tilde{x}) = y)].$$

Let $\varepsilon_i : \mathbb{N} \mapsto [0, 1]$ be a monotonically increasing function. Then, the function ψ is an $\varepsilon_i(\cdot)$–OWF if for all adversaries $\mathcal{A}$ it holds that $\mathbf{Adv}^{\mathsf{OWF}}_{\mathcal{A}_\tau,\psi} \le \varepsilon_i(\tau)$.

We then extend this definition to a family of functions as follows:

Definition 6 **($\varepsilon(\cdot)$–OWFF).** *Let $\varphi : \mathbb{N} \mapsto \big(\mathcal{X} \mapsto \mathcal{Y}\big)$ and $\varepsilon : \mathbb{N} \mapsto \big(\mathbb{N} \mapsto [0, 1]\big)$ be function families. We say φ is an $\varepsilon(\cdot)$–OWFF if for all $i \in \mathbb{N}$ the function $\varphi_i : \mathcal{X} \mapsto \mathcal{Y}$ is an $\varepsilon_i(\cdot)$–OWF.*

THE GENERIC CONSTRUCTION. Our generic construction is based on the method of Juels and Brainard [14]. Most client puzzle constructions based on one way functions, such as the discrete log based scheme of [27], and the RSA based scheme of [10], can be described in this manner with some minor modifications. So, our generic construction pins down sufficient assumptions on the building blocks that imply security of the resulting puzzle. We let $k \in \mathbb{N}$ then let $F : \mathcal{K} \times D \mapsto \mathcal{X}$ where $|\mathcal{X}| \ge |\mathcal{K}| = 2^k$ be a function family indexed by elements of $\mathcal{K}$. The domain D of F_s is 3-tuples of the form $\mathbb{N} \times \{0, 1\}^* \times \{0, 1\}^k \in \{0, 1\}^*$. We write $F_s((\cdot, \cdot, \cdot))$ when we want to specify the exact encoding of an element of D explicitly as an input to F_s. We further let $\varphi : \mathbb{N} \mapsto (\mathcal{X} \mapsto \mathcal{Y})$ be a family of functions indexed by Q. We assume there is a polynomial time algorithm to compute φ_Q for each value of Q and input. The various algorithms in the scheme are then as follows:

$\mathsf{Setup}(1^k)$. The various spaces are chosen; $\mathsf{sSpace} \leftarrow \mathcal{K}$, $\mathsf{QSpace} \leftarrow \mathbb{N}$, $\mathsf{strSpace} \leftarrow \{0,1\}^*$, $\mathsf{solnSpace} \leftarrow \mathcal{X}$ and $\mathsf{puzSpace} \leftarrow \mathsf{QSpace} \times \mathsf{strSpace} \times \{0,1\}^k \times \mathcal{Y}$. The parameter Π is assigned to be the polynomial time algorithm to compute φ_Q for all $Q \in \mathsf{QSpace}$ and $x \in \mathcal{X}$. Finally, the value s is chosen as $s \overset{\$}{\leftarrow} \mathsf{sSpace}$ and the tuple params constructed then output.

$\mathsf{GenPuz}(s, Q, \mathsf{str})$. A nonce is selected $n_S \overset{\$}{\leftarrow} \{0,1\}^k$. Next x is computed as $x \leftarrow F_s(Q, \mathsf{str}, n_S)$. The value $y \in \mathcal{Y}$ is computed as $y \leftarrow \varphi_Q(x)$ and the puzzle assigned to be $\mathsf{puz} = (Q, \mathsf{str}, n_S, y)$ and output.

$\mathsf{FindSoln}(\mathsf{puz}, \tau)$. While this algorithm is within the allowed number of clock cycles of execution it randomly samples elements from the set of possible solutions without replacement and for each potential preimage $x' \in \mathcal{X}$ computes $y' \leftarrow \varphi_Q(x')$. If $y' = y$ this outputs x' then halts and otherwise continues with random sampling. If this algorithm reaches the last clock cycle of execution then it outputs a random element of the remaining unsampled preimage space. The set of possible solutions is generally a subset of $\mathcal{X}$ that is defined by the value y of size dependent upon Q in some manner; the details of how the size varies depends upon the function family φ.

$\mathsf{VerAuth}(s, \mathsf{puz}')$. For a puzzle $\mathsf{puz}' = (Q', \mathsf{str}', n'_S, y')$ this computes x' as $x' \leftarrow F_s(Q', \mathsf{str}', n'_S)$ then $y \leftarrow \varphi_Q(x')$. If $y' = y$ this outputs **true** and otherwise outputs **false**.

$\mathsf{VerSoln}(\mathsf{puz}', \mathsf{soln}')$. Given a potential solution $\mathsf{soln}' = x'$ this checks if $\varphi_Q(x') = y$ and if so outputs **true** and otherwise outputs **false**.

We use the notation $\mathsf{CPuz} = \mathsf{PROWF}(F; \varphi)$ for the generic construction in this manner. The construction is summarized in Figure 3.

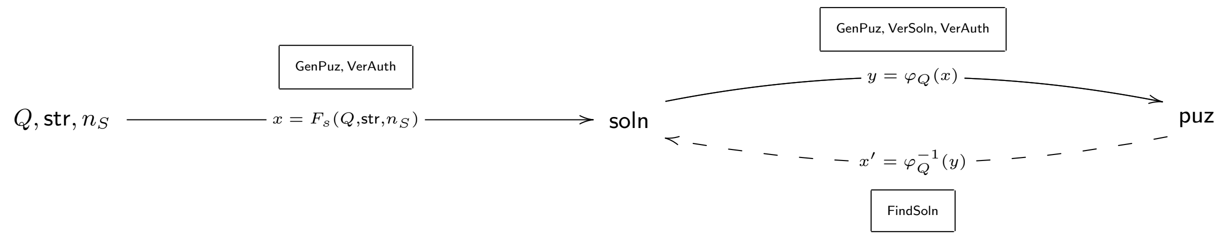


Fig. 3. Solid arrows are actions performed by a generator and dashed ones by a solver. The lists of algorithms above/below arrows imply the actions are performed as part of these algorithms. The details of how each action is used in the given algorithm are given in the full description.

Remark 1. In the definition of an $\varepsilon(\cdot)$–OWF we specify the domain $\mathcal{X}$ is fixed and finite but do not specify the exact size or shape of this; in our generic construction this is set to be the output space of some PRF.

Remark 2. The exact specification of the $\mathsf{FindSoln}$ algorithm is not important for our theorems and proofs nor is it unique. Indeed, other techniques such as exhaustive search may even be faster than the algorithm given. The important point is such an algorithm exists and can be described.

Remark 3. The domain D of F is given as 3 tuples of the form $\mathbb{N} \times \{0,1\}^* \times \{0,1\}^k$ which is the same as $\{0,1\}^*$. However, we will always construct elements of D from a given tuple rather than taking a bit string and encoding it as an element of D. Hence we do not refer to this as a uniquely recoverable encoding on D.

Remark 4. In reality the variable n_S need not be sampled at random; it just has to be a nonce and could be instantiated with, for example, a counter. We specify uniform sampling from the domain of n_S since it makes our proofs simpler and easier to follow.

Remark 5. Our generic construction is similar to the Juels and Brainard scheme [14] but avoids the forgeability problems by including the hardness parameter Q in the input to F.

Remark 6. Finally, we remark that the generic construction where the PRF function is replaced by a MAC is not necessarily secure. Indeed, one-wayness of the generic construction is guaranteed as long as the one-way function is applied to randomly chosen bit-string. While this property is ensured through the use of a pseudo-random function, it is does not always hold for a MAC. For example, using an artificial MAC in which most (say the first 3/4) of the output bits are constant, and a one-way function which discards only and precisely those bits, leads to an yield an insecure construction.

The following theorems capture the security of this generic construction. Their proofs can be found in Appendix B.1 and Appendix B.2 respectively.

Theorem 1. *Let F be a* PRF *family and φ a family of functions as described above such that for each value of Q and for all $y \in \mathcal{Y}$ we have $|\varphi_Q^{-1}(y)|/|\mathcal{X}| \leq 1/2^k$, where k is the security parameter. Then the client puzzle* $\mathsf{CPuz} = \mathsf{PROWF}(F;\varphi)$ *is* UF *secure.*

The condition that $|\varphi_Q^{-1}(y)|/|\mathcal{X}| \leq 1/2^k$ for all values of Q and for all $y \in \mathcal{Y}$ in Theorem 1 prevents the following attack. An adversary may try to forge a puzzle instance by choosing a value for y then selecting a triple (Q, str, n_S). If the value y has a large number of preimages then the adversary may win without attacking the pseudorandomness of F at all.

Theorem 2. *Let F be a $\nu(\cdot)$–*PRFF *family for the function family $\nu(\cdot)$, φ an $\varepsilon(\cdot)$–*OWFF *for the function family $\varepsilon(\cdot)$ and* $\mathsf{CPuz} = \mathsf{PROWF}(F;\varphi)$. *Then* $\mathsf{PROWF}(F;\varphi)$ *is $\gamma(\cdot)$–*DIFF *where $\gamma_{k,Q}(\tau) = 2\cdot\nu_k(\tau+\tau^0) + \big(1+\tau/(2^k-\tau)\big)\cdot \varepsilon_Q(\tau+\tau^1)$ and $\tau^o, \tau^1 \in \mathbb{N}$ are some constants.*

An adversary may try to solve puzzles by either computing the value $F_s(Q, \mathsf{str}, n_S)$ for an unknown value of s or by computing a preimage of φ_Q for the value y provided. The function ν_k in Theorem 2 captures that computing F_s for an unknown value of s should not be easy; the function F needs to be a good PRF. Intuitively, k should be chosen to be large enough that it is easier to compute a preimage of y under φ_Q than computing the corresponding value $F_s(Q, \mathsf{str}, n_S)$.

Impact on Practical Implementations of Puzzles

The use of cryptographic puzzle schemes where puzzle creation is expensive, immediately leads to DoS attacks. From this perspective the use of a provably secure PRF would most likely be avoided by practical implementations. We now present an efficient implementation of our generic construction based on cryptographic hash functions. The scheme follows by instantiating the two components of our generic construction. We obtain essentially a modified Juels and Brainard scheme that incorporates the defence against the attack that we present in Section 5.

Given a hash function $H : \{0,1\}^* \mapsto \{0,1\}^m$ a standard construction for a PRF family F is as follows. Key generation selects a random string $s \xleftarrow{\$} \{0,1\}^k$ where k is the security parameter. Function application is defined by $F_s(x) = H(s||x)$ for any $x \in \{0,1\}^*$ Furthermore, given a hash function $G : \{0,1\}^* \to \{0,1\}^n$ we define the function family φ of functions $\varphi_Q : \{0,1\}^m \to \{0,1\}^{m-Q} \times \{0,1\}^n$ by $\varphi_Q(x) = (x\langle Q+1,\ m\rangle, G(x))$. In Appendix C we prove that in the random oracle model, F is a $\nu_k(\cdot)$–PRFF function, for some some function family ν with $\nu_k(\tau) \leq \frac{m}{2^k}$ and that φ is $\varepsilon(\cdot)$–OWFF for some function family ε with $\varepsilon(\tau) \leq \tau/2^m + \tau/(2^{m-Q})$. Concrete bounds for the security of our construction follow by instantiating the bounds in Theorems 1 and 2.

References

1. W. Aiello, S. M. Bellovin, M. Blaze, R. Canetti, J. Ioannidis, A. D. Kermoytis and O. Reingold. Just Fast Keying: Key Agreement In A Hostile Internet. In *ACM Transactions on Information and System Security* Vol 4, No. 2, 1–30, 2004.
2. T. Aura, P. Nikander and J. Leiwo. DoS-Resistant Authentication with Client Puzzles. In *Security Protocols Workshop*. Springer-Verlag LNCS 2133, 170–181, 2001.
3. B. Barak and M. Mahmoody–Ghidary. Merkle Puzzles are Optimal. Cryptology ePrint archive, report 2008/032, 2008.

4. E. Biham, Y. J. Goren and Y. Ishai. Basing Weak Public-Key Cryptoraphy on Strong One-Way Functions In *Theory of Cryptography Conference*. Springer-Verlag LNCS 4948, 55–72, 2008.
5. R. Canetti, S. Halevi and M. Steiner. Hardness Amplification of Weakly Verifiable Puzzles. In *Proceedings of Theory of Cryptography Conference – TCC 2005.* Springer-Verlag LNCS 3378, 17–33, 2005.
6. L. Chen and W. Mao. An Auditable Metering Scheme for Web Advertisement Applications. In *Proceedings of the 4th International Conference on Information Security*, Springer-Verlag LNCS 2200, 475–485, 2001.
7. C. Douligeris and A. Mitrokotsa. DDoS Attacks and Defence mechanisms: Classification and State–of–the–Art. In *Computer Networks* Vol 44, Issue 5, 643–666, 2004.
8. C. Dwork and M. Naor. Pricing via Processing or Combatting Junk Email. In *Advances in Cryptology – CRYPTO 1992* Springer-Verlag LNCS Vol. 740, 139–147, 1992.
9. W. Eddy. *TCP SYN Flooding Attacks and Common Mitigations.* RFC 4987, August 2007.
10. Y. Gao. Efficient Trapdoor-Based Client Puzzle System Against DoS Attacks. MSc Thesis, University of Wollongong, Computer Science Department. 2005.
11. B. Groza and Dorina Petrica. On Chained Cryptographic Puzzles. In *3rd Romanian-Hungarian Joint Symposium on Applied Computational Intelligence (SACI).* 25–26, 2006.
12. R. Impagliazzo and S. Rudich. Limits on the Provable Consequences of One–Way Permutations. In *Proceedings of the 21st Annual ACM Symposium on the Theory of Computing – STOC 1989.* ACM, pages 44–61, 1989.
13. M. Jakobsson and A. Juels. Proofs of Work and Bread Pudding Protocols. In *Proceedings of the IFIP TC6/TC11 Joint Working Conference on Secure Information Networks: Communications and Multimedia Security.* IFIP Conference Proceedings Vol. 152, 258–272, 1999.
14. A. Juels and J. Brainard. Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks In *Proceedings of the 1999 ISOC Network and Distributed System Security Symposium.* 151–165, 1999.
15. D. Karig and R. Lee. Remote Denial of Service Attacks and Countermeasures Princeton University Department of Electrical Engineering Technical Report CE–L2001–002. 2001.
16. C. Meadows. A Formal Framework and Evaluation Method for Network Denial of Service. In *Proceedings of the 12th Computer Security Foundations Workshop* IEEE Computer Society Press, 4–13, 1999.
17. C. Meadows A Cost–Based Framework for Analysis of Denial of Service in Networks. In *Journal of Computer Security* Vol 9, Issues 1–2, 143–164, 2001.
18. R. Merkle. Secure Communications Over Insecure Channels. In *Communications of the ACM.* Vol. 21, Issue 4, 294–299, 1978.
19. J. Mirkovic, J. Martin and P. Reiher. A Taxonomy of DDoS Attack and DDoS Defense Mechanisms. In *ACM SIGCOMM Computer Communication Review* Vol 34, Issue 2, 39–53, 2004.
20. R. Moskowitz, P. Nikander, P. Jokela and T. Henderson. *Host Identity Protocol.* Internet Draft, October 2007.
21. R. L. Rivest, A. Shamir and D. Wagner. Time-lock Puzzles and Timed-release Crypto. Massachusetts Institute of Technology Technical Report TR-684. 1996.
22. P. Rogaway. Formalizing Human Ignorance. In *Progress in Cryptology – VIETCRYPT 2006.* Springer-Verlag LNCS Vol. 4341, 211–228, 2006.
23. P. Schaller, S. Capkun and D. Basin. BAP: Broadcast Authentication Using Cryptographic Puzzles. In *Applied Cryptography and Network Security.* Springer-Verlag LNCS Vol. 4521, 401–419, 2007.
24. J. Smith, J. M. González–Nieto and C. Boyd Modelling Denial of Service Attacks on JFK with Meadows's Cost–Based Framework. In *Proceedings of the 2006 Australasian workshop on Grid computing and e–research* Vol 54, 125–134, 2006.
25. S. Specht and R. Lee. Distributed Denial of Service: Taxonomies of Attacks, Tools, and Countermeasures. In *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems, 2004 International Workshop Security in Parallel and Distributed Systems* 543–550, 2004.
26. S. Tritilanunt, C. Boyd, E. Foo and J. M. González–Nieto. Toward Non-Parallelizable Client Puzzles. In *Cryptology and Network Security – CANS 2007.* LNCS Vol. 4896, 247–264, 2007.
27. B. Waters, A. Juels, J. A. Halderman and E. W. Felten. New Client Puzzle Outsourcing Techniques for DoS Resistance. In *Proceedings of the 11th ACM Conference on Computer and Communication Security – CCS* ACM Press, 246–256, 2004.

A Notation

A function $\epsilon(t)$ is said to be *negligible* in the parameter t if $\forall\ c \geq \mathbb{Z}_{>0}\ \exists\ t_c \in \mathbb{R}_{>0}$ such that $\forall\ t > t_c, \epsilon(t) < t^{-c}$.

If S is any set then we denote the action of sampling an element from S uniformly at random and assigning the result to the variable x as $x \overset{\$}{\leftarrow} S$.

We let $\mathrm{d.p.t.}$ denote deterministic polynomial time and $\mathrm{p.p.t.}$ probabilistic polynomial time. If F is some function we use the notation $x \leftarrow F(y_1, \ldots, y_n)$ for the process of obtaining x by running F on inputs $y_1, y_2, \ldots, y_n$. If A is some algorithm we use the notation $x \leftarrow A^{\mathcal{O}_1(\cdot), \mathcal{O}_2(\cdot)}(y_1, \ldots, y_n)$ for the process of obtaining x by running A on inputs $y_1, y_2, \ldots, y_n$ with access to oracles $\mathcal{O}_1(\cdot)$ and $\mathcal{O}_2(\cdot)$. If x is some variable then we also use this for running A or F with the given inputs and then assigning the output to x.

B Proofs of Theorems

B.1 Proof of Theorem 1

Proof. Consider some adversary $\mathcal{A}$ against the UF security of $\mathsf{PROWF}(F; \varphi)$. We define games $\mathbf{G}_0$ and $\mathbf{G}_1$ by $\mathbf{G}_0 = \mathsf{Exec}^{\mathsf{UF}}_{\mathcal{A},\mathsf{CPuz}}(k)$ and $\mathbf{G}_1 = \mathsf{Exec}^{\mathsf{UF}}_{\mathcal{A},\mathsf{CPuz}'}(k)$ where the schemes involved are defined by $\mathsf{CPuz} = \mathsf{PROWF}(F; \varphi)$, $\mathsf{CPuz}' = \mathsf{PROWF}(\mathcal{R}; \varphi)$ and $\mathcal{R}$ is some function selected at random from the set of functions $\mathsf{Func}(D, \mathcal{X})$. In other words, in game $\mathbf{G}_0$ the adversary tries to break our construction whereas in game $\mathbf{G}_1$ the adversary works against an idealized version of our construction, where the pseudorandom function F had been replaced with a truly random function. We now argue that if the adversary $\mathcal{A}$ sees a difference between the two games, then he breaks the security of the pseudorandom function. We do this by constructing an adversary $\mathcal{B}$ such that:

$$\Big|\Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_0\big] - \Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_1\big]\Big| = 2 \cdot \mathbf{Adv}^{\mathsf{PRF}}_{\mathcal{B},F}(k).$$

Since $\Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_0\big] = \mathbf{Adv}^{\mathsf{UF}}_{\mathcal{A},\mathsf{CPuz}}(k)$, the theorem follows by bounding $\Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_1\big]$.

We now describe the details of the adversary $\mathcal{B}$. For $b = 0$, at the start of the game $\mathsf{Exec}^{\mathsf{PRF},b}_{\mathcal{B},F}(k)$, the adversary $\mathcal{B}$ is given black box access to a function $F_{s^\dagger}$ from the family F with an unknown value of key $s^\dagger$. For $b = 1$ the adversary $\mathcal{B}$ is given black box access to a function $\mathcal{R}$ chosen randomly from the set of functions $\mathsf{Func}(D, \mathcal{X})$. The input strings to $\mathcal{R}$ will have the same syntax as those to the members of $F_{s^\dagger}$. The adversary $\mathcal{B}$ then acts as a challenger to the adversary $\mathcal{A}$. It does this as follows:

(1) The adversary $\mathcal{B}$ first runs Setup on input 1^k to obtain a pair (params, s) and starts the adversary $\mathcal{A}$ on input params. Throughout the simulation $\mathcal{B}$ does not use the value of s but instead answers queries made by $\mathcal{A}$ using its black box access to either $F_{s^\dagger}$ or $\mathcal{R}$ depending on the value of b.

(2) The adversary $\mathcal{A}$ will then start to ask $\mathsf{CreatePuz}(Q, \mathsf{str})$ and $\mathsf{CheckPuz}(\mathsf{puz})$ queries which $\mathcal{B}$ answers as follows:

- $\mathsf{CreatePuz}(Q, \mathsf{str})$ queries. For this the adversary $\mathcal{B}$ selects a random nonce $n_S \xleftarrow{\$} \{0,1\}^k$ then computes x by querying whichever function it has black box access to on input (Q, str, n_S). Next $\mathcal{B}$ computes y as the output of $\varphi_Q(x)$ and replies to $\mathcal{A}$ with the puzzle $\mathsf{puz} = (Q, \mathsf{str}, n_S, y)$. The adversary $\mathcal{B}$ also keeps a list of all puzzles output to $\mathcal{A}$.
- $\mathsf{CheckPuz}(\mathsf{puz})$ queries. For a given $\mathsf{puz} = (Q, \mathsf{str}, n_S, y)$ if puz was output by $\mathcal{B}$ in response to a $\mathsf{CreatePuz}$ query then $\mathcal{B}$ outputs $\mathbf{true}$. Otherwise $\mathcal{B}$ queries whichever function it has black box access to on inputs (Q, str, n_S) and receives a response $\overline{x}$. Next $\mathcal{B}$ computes $\overline{y}$ as the output of $\varphi_Q(\overline{x})$. If $y = \overline{y}$ then $\mathcal{B}$ outputs 1 in its game against $\mathcal{C}$ and terminates and otherwise responds with $\mathbf{false}$.

(3) If $\mathcal{B}$ has not terminated in response to a $\mathsf{CheckPuz}$ query then eventually $\mathcal{A}$ will terminate. The adversary $\mathcal{B}$ then selects a bit at random, outputs this in its game against $\mathcal{C}$ and terminates.

If $b = 0$ then the above game between $\mathcal{B}$ and $\mathcal{A}$ will be exactly the game $\mathbf{G}_0$ and if $b = 1$ we get the game $\mathbf{G}_1$. Hence we have

$$\Pr\Big[\mathsf{Exec}^{\mathsf{PRF},b}_{\mathcal{B},F}(k) = 1\Big] = \Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_b\big] + \frac{1}{2}\Big(1 - \Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_b\big]\Big).$$

As a result we get

$$2 \cdot \mathbf{Adv}^{\mathsf{PRF}}_{\mathcal{B},F}(k) = \Big|\Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_0\big] - \Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_1\big]\Big|.$$

We now bound $\Pr[\mathcal{A} \text{ wins } \mathbf{G}_1]$. If $\mathcal{A}$ wins $\mathbf{G}_1$ we let $\mathsf{puz}^\dagger = (Q^\dagger, \mathsf{str}^\dagger, n_S^\dagger, y^\dagger)$ be the puzzle on which $\mathcal{A}$ wins the game, let $x^\dagger = \mathcal{R}(Q^\dagger, \mathsf{str}^\dagger, n_S^\dagger)$ and let $\mathsf{view}(\mathcal{A})$ be the transcript of $\mathcal{A}$; everything that adversary $\mathcal{A}$ sees during the game. Since $\mathcal{A}$ wins it must be the case that $\mathsf{puz}^\dagger$ was not output in response to a $\mathsf{CreatePuz}$ query. Since $\varphi_{Q^\dagger}$ is deterministic $(Q^\dagger, \mathsf{str}^\dagger, n_S^\dagger)$ cannot have been output by $\mathcal{R}$ during a $\mathsf{CreatePuz}$ query; otherwise $\mathsf{puz}^\dagger$ would have been output in response to this query. Furthermore, since $\mathcal{R}$ is a random function, $y^\dagger$ must have been chosen by $\mathcal{A}$ independently of $x^\dagger$ given $\mathsf{view}(\mathcal{A})$. Since $x^\dagger$ is independent from $\mathsf{view}(\mathcal{A})$, the probability that the adversary wins in the game is:

$$\Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_1\big] = \Pr\Big[\varphi_{Q^\dagger}(x^\dagger) = y^\dagger\Big]$$

where the second probability is over the choice of $x^\dagger$ in $\mathcal{X}$. In other words, we have that:

$$\Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_1\big] = \frac{\Big|\varphi_{Q^\dagger}^{-1}(y^\dagger)\Big|}{|\mathcal{X}|}$$

We therefore have:

$$\begin{aligned}
\mathbf{Adv}^{\mathsf{UF}}_{\mathcal{A},\mathsf{CPuz}}(k) &= \Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_0\big] \\
&= \Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_0\big] + \Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_1\big] - \Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_1\big] \\
&\leq \big|\Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_0\big] - \Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_1\big]\big| + \Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_1\big] \\
&= 2 \cdot \mathbf{Adv}^{\mathsf{PRF}}_{\mathcal{B},F}(k) + \frac{|\varphi_{Q^\dagger}^{-1}(y^\dagger)|}{|\mathcal{X}|} \\
&\leq 2 \cdot \mathbf{Adv}^{\mathsf{PRF}}_{\mathcal{B},F}(k) + \frac{1}{2^k}.
\end{aligned}$$

□

B.2 Proof of Theorem 2

Proof. We prove this theorem using a technique similar to the proof of Theorem 1. Again we define two games $\mathbf{G}_0$ and $\mathbf{G}_1$ but this time have $\mathbf{G}_0 = \mathsf{Exec}^{\mathsf{DIFF}}_{\mathcal{A},\mathsf{CPuz}}(k)$ and $\mathbf{G}_1 = \mathsf{Exec}^{\mathsf{DIFF}}_{\mathcal{A},\mathsf{CPuz}'}(k)$ where $\mathsf{CPuz} = \mathsf{PROWF}(F;\varphi)$, $\mathsf{CPuz}' = \mathsf{PROWF}(\mathcal{R};\varphi)$ and $\mathcal{R}$ is some random function selected from the set of functions $\mathsf{Func}(D,\mathcal{X})$. We again argue that if an adversary $\mathcal{A}_\tau$ sees a difference between the two games then he breaks the concrete security of the pseudorandom function family. We do this by constructing an adversary $\mathcal{B}$ such that

$$\Big|\Pr\big[\mathcal{A}_\tau \text{ wins } \mathbf{G}_0\big] - \Pr\big[\mathcal{A}_\tau \text{ wins } \mathbf{G}_1\big]\Big| = 2 \cdot \mathbf{Adv}^{\mathsf{PRF}}_{\mathcal{B}_{\tau+\tau^0},F}(k).$$

The main difference with this proof and that of Theorem 1 is for the game $\mathbf{G}_1$ we construct an adversary $\mathcal{C}$ against φ_Q such that $\mathbf{Adv}_{\mathcal{C}_{\tau+\tau^1},\varphi_Q} = (1 + q/(2^k - q)) \cdot \Pr\big[\mathcal{A}_\tau \text{ wins } \mathbf{G}_1\big]$ where q is the number of $\mathsf{CreatePuzSoln}$ queries made by $\mathcal{A}$ on a certain string. Then, since $\Pr\big[\mathcal{A}_\tau \text{ wins } \mathbf{G}_0\big] = \mathbf{Succ}^{Q,\mathsf{DIFF}}_{\mathcal{A}_\tau,\mathsf{CPuz}}(k)$, $\mathbf{Adv}^{\mathsf{PRF}}_{\mathcal{B}_{\tau+\tau^0},F}(k) \leq \nu_k(\tau + \tau^0)$ and $\mathbf{Adv}_{\mathcal{C}_{\tau+\tau^1},\varphi_Q} \leq \varepsilon_Q(\tau + \tau^1)$ the theorem follows.

We now describe the details of the adversary $\mathcal{B}$. For $b = 0$, at the start of the game $\mathsf{Exec}^{\mathsf{PRF},b}_{\mathcal{B},F}(k)$, the adversary $\mathcal{B}$ is given black box access to a function $F_{s^\dagger}$ from the family F with an unknown value of key $s^\dagger$. For $b = 1$ the adversary $\mathcal{B}$ is given black box access to a function $\mathcal{R}$ chosen randomly from the set of functions $\mathsf{Func}(D,\mathcal{X})$. The input strings to $\mathcal{R}$ will have the same syntax as those to the members of $F_{s^\dagger}$. The adversary $\mathcal{B}$ then acts as a challenger to the adversary $\mathcal{A}$. It does this as follows:

(1) The adversary $\mathcal{B}$ first runs Setup on input 1^k to obtain a pair (params, s) and starts the adversary $\mathcal{A}$ on input params. Throughout the simulation $\mathcal{B}$ does not use the value of s but instead answers queries made by $\mathcal{A}$ using its black box access to either $F_{s^\dagger}$ or $\mathcal{R}$ depending on the value of b.

(2) The adversary $\mathcal{A}$ will then start to ask $\mathsf{CreatePuzSoln}(\mathsf{str})$ queries. To answer each $\mathcal{B}$ selects a random nonce $n_S \xleftarrow{\$} \{0,1\}^k$ then computes x by querying whichever function it has black box access to on input (Q, str, n_S). Next $\mathcal{B}$ computes $y \leftarrow \varphi_Q(x)$ then assigns $\mathsf{puz} \leftarrow (Q, \mathsf{str}, n_S, y)$ and $\mathsf{soln} \leftarrow x$ and passes $(\mathsf{puz}, \mathsf{soln})$ to $\mathcal{A}$.

(3) Eventually $\mathcal{A}$ will make a single $\mathsf{Test}(\mathsf{str}^*)$ query. The adversary $\mathcal{B}$ then generates a challenge puzzle in the same manner as when answering $\mathsf{CreatePuzSoln}$ queries. We denote the challenge puzzle as $\mathsf{puz}^* = (Q, \mathsf{str}^*, n_S^*, y^*)$.

(4) The adversary $\mathcal{A}$ will then continue to ask $\mathsf{CreatePuzSoln}(\mathsf{str})$ queries which $\mathcal{B}$ answers as before.

(5) After τ clock cycles $\mathcal{A}$ terminates outputting a potential solution $\mathsf{soln}^* = x^*$. If $\mathcal{A}$ wins, i.e. if $\mathsf{VerSoln}(\mathsf{puz}^*, \mathsf{soln}^*) = \mathbf{true}$, then $\mathcal{B}$ outputs 1 and terminates and otherwise outputs a random bit and terminates.

By the way we construct $\mathcal{B}$, if $b = 0$ then the above game between that $\mathcal{B}$ simulates for $\mathcal{A}$ will be exactly the game $\mathbf{G}_0$ and if $b = 1$ the game $\mathcal{B}$ simulates for $\mathcal{A}$ will be exactly the game $\mathbf{G}_1$. Hence we have

$$\Pr\big[\mathsf{Exec}^{\mathsf{PRF},b}_{\mathcal{B},F}(k) = 1\big] = \Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_b\big] + \frac{1}{2}\Big(1 - \Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_b\big]\Big).$$

As a result we get

$$2 \cdot \mathbf{Adv}^{\mathsf{PRF}}_{\mathcal{B},F}(k) = \Big|\Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_0\big] - \Pr\big[\mathcal{A} \text{ wins } \mathbf{G}_1\big]\Big|.$$

The constant τ^0 is the time $\mathcal{B}$ takes on top of the time for the game; the simulation will take some additional clock cycles since $\mathcal{B}$ needs to make its own queries in addition to the time taken by $\mathcal{A}$ for the game. We refer to this as the "overhead" time. Since we define the running time of $\mathcal{A}$ to be the running time of all the experiment $\mathsf{Exec}^{Q,\mathsf{DIFF}}_{\mathcal{A},\mathsf{CPuz}}(k)$ we have that the running time of $\mathcal{B}$ will be $\tau + \tau^o$.

We now construct the adversary $\mathcal{C}$ against φ_Q. At the start of its game $\mathcal{C}$ is given an element $y^\dagger \in \mathcal{Y}$. Throughout the game $\mathcal{C}$ will simulate a random function $\mathcal{R}$ from $\mathsf{Func}(D, \mathcal{X})$ by selecting a random element from $\mathcal{X}$ for each new value of D queried and maintaining a list of queries to answer consistently for previously queried values of D. The adversary $\mathcal{C}$ then acts as a challenger to $\mathcal{A}$ in a game $\mathbf{G}_1$ as follows:

(1) The adversary $\mathcal{C}$ first runs Setup on input 1^k to obtain a pair (params, s) and starts the adversary $\mathcal{A}$ on input params.

(2) The adversary $\mathcal{A}$ will then start to ask $\mathsf{CreatePuzSoln}(\mathsf{str})$ queries. To answer each $\mathcal{C}$ selects a random nonce $n_S \xleftarrow{\$} \{0,1\}^k$ then computes $x \leftarrow \mathcal{R}(Q, \mathsf{str}, n_S)$. Next $\mathcal{C}$ computes $y \leftarrow \varphi_Q(x)$ then assigns $\mathsf{puz} \leftarrow (Q, \mathsf{str}, n_S, y)$ and $\mathsf{soln} \leftarrow x$ and passes $(\mathsf{puz}, \mathsf{soln})$ to $\mathcal{A}$. We assume the adversary $\mathcal{C}$ keeps a record $\mathcal{Q}$ of all such queries made by $\mathcal{A}$.

(3) Eventually $\mathcal{A}$ will make a single $\mathsf{Test}(\mathsf{str}^\dagger)$ query. The adversary $\mathcal{C}$ then generates a challenge puzzle by selecting a random nonce $n_S^\dagger \xleftarrow{\$} \{0,1\}^k$ then assigns the challenge puzzle to be $\mathsf{puz}^\dagger = (Q, \mathsf{str}^\dagger, n_S^\dagger, y^\dagger)$. If there exists a query on the list $\mathcal{Q}$ so that the puzzle output is $\mathsf{puz}' = (Q, \mathsf{str}^\dagger, n_S^\dagger, *)$ then $\mathcal{C}$ terminates and otherwise outputs $\mathsf{puz}^\dagger$ to $\mathcal{A}$.

(4) The adversary $\mathcal{A}$ will then continue to ask $\mathsf{CreatePuzSoln}(\mathsf{str})$ queries. Algorithm $\mathcal{C}$ computes a puzzle and solution pair $(\mathsf{puz}, \mathsf{soln})$ as before. In the case where $\mathsf{str} = \mathsf{str}^\dagger$ then if there is an entry on $\mathcal{Q}$ so that the puzzle output is $\mathsf{puz}' = (Q, \mathsf{str}^\dagger, n_S^\dagger, *)$ then $\mathcal{C}$ aborts and otherwise outputs $(\mathsf{puz}, \mathsf{soln})$ to $\mathcal{A}$.

(5) After τ clock cycles $\mathcal{A}$ terminates outputting a potential solution $\mathsf{soln}^\dagger = x^\dagger$. The adversary $\mathcal{C}$ then outputs $x^\dagger$ in its game against φ_Q and terminates.

In the above simulation provided by $\mathcal{C}$, the function $\mathcal{R}$ that $\mathcal{C}$ simulates is a random one. As a result the only way $\mathcal{A}$ could find a correct solution without $\mathcal{C}$ aborting, i.e. without $\mathcal{C}$ computing $\mathcal{R}(Q, \mathsf{str}^\dagger, n_S^\dagger)$ at some point, would be for $\mathcal{A}$ to invert the function φ_Q.

To construct the advantage statements for this game we let E denote the event that $\mathcal{C}$ aborts. Since the choice of each nonce n_S is random and independent of the behavior of $\mathcal{A}$ the event that $\mathcal{A}$ wins $\mathbf{G}_1$ is independent of whether $\mathcal{C}$ aborts or not. As a result we have

$$\mathbf{Adv}^{\mathsf{OWF}}_{\mathcal{C}_{\tau+\tau^1},\varphi_Q} = \Pr\big[\mathcal{A}_\tau \text{ wins } \mathbf{G}_1\big] \cdot \Pr\big[\neg E\big].$$

The adversary $\mathcal{C}$ will abort only if a query $\mathsf{CreatePuzSoln}(\mathsf{str}^{\dagger})$ is answered using $n_S^{\dagger}$ at some point during the game. This will occur with probability $q/2^k$ where q is the number of $\mathsf{CreatePuzSoln}(\mathsf{str}^{\dagger})$ queries made by $\mathcal{A}$ hence $\Pr[\neg E] = 1 - q/2^k$ and we can write

$$\Pr[\mathcal{A}_\tau \text{ wins } \mathbf{G}_1] = \left(1 + \frac{q}{2^k - q}\right) \cdot \mathbf{Adv}^{\mathsf{OWF}}_{\mathcal{C}_{\tau+\tau^1}, \varphi_Q}(k).$$

The constant τ^1 captures the difference, in clock cycles of execution, between an actual $\mathbf{G}_1$ game and the simulation $\mathcal{C}$ provides in the same way as τ^0 is used.

As a result we get

$$\begin{aligned}
\mathbf{Succ}^{Q,\mathsf{DIFF}}_{\mathcal{A}_\tau,\mathsf{CPuz}}(k) &= \Pr[\mathcal{A}_\tau \text{ wins } \mathbf{G}_0] \\
&= \Pr[\mathcal{A}_\tau \text{ wins } \mathbf{G}_0] + \Pr[\mathcal{A}_\tau \text{ wins } \mathbf{G}_1] - \Pr[\mathcal{A}_\tau \text{ wins } \mathbf{G}_1] \\
&\le \left|\Pr[\mathcal{A}_\tau \text{ wins } \mathbf{G}_0] - \Pr[\mathcal{A}_\tau \text{ wins } \mathbf{G}_1]\right| + \Pr[\mathcal{A}_\tau \text{ wins } \mathbf{G}_1] \\
&= 2 \cdot \mathbf{Adv}^{\mathsf{PRF}}_{\mathcal{B}_{\tau+\tau^0},F}(k) + \Pr[\mathcal{A}_\tau \text{ wins } \mathbf{G}_1] \\
&= 2 \cdot \mathbf{Adv}^{\mathsf{PRF}}_{\mathcal{B}_{\tau+\tau^0},F}(k) + \left(1 + \frac{q}{2^k - q}\right) \cdot \mathbf{Adv}^{\mathsf{OWF}}_{\mathcal{C}_{\tau+\tau^1},\varphi_Q} \\
&\le 2 \cdot \nu_k(\tau + \tau^0) + \left(1 + \frac{q}{2^k - q}\right) \cdot \varepsilon_Q(\tau + \tau^1).
\end{aligned}$$

Finally, since the running time of $\mathcal{A}$ is always at most τ we can bound the number of queries q by τ giving the desired result. □

C A More Detailed Analysis of our Random Oracle Based Instantiation

In this section we derive the concrete security bounds for the PRF family and the one-way function family implemented with random oracles.

Consider the game $\mathsf{Exec}^{\mathsf{PRF},0}_{\mathcal{A},F}$ where H is used to instantiate F. In this case the adversary will be able to query F in a black box way, meaning it can make a query on some δ and receive $H(s^* \mid\mid \delta)$ for some unknown value s^*. The adversary will also have access to the random oracle H which it can query on any input it likes. We define the event E to be the event that the adversary makes a query to H (without using black box access to F_{s^*}) of the form $H(s^* \mid\mid \delta)$. If event E occurs then $\mathcal{A}$ could make a query on δ using its black box access to F and would get the same value back as for the event E. We then have

$$\Pr\left[\mathsf{Exec}^{\mathsf{PRF},0}_{\mathcal{A}_\tau,F}(k) = 1\right] = \Pr\left[\mathsf{Exec}^{\mathsf{PRF},0}_{\mathcal{A},F}(k) = 1 \mid E\right] \cdot \Pr[E] + \Pr\left[\mathsf{Exec}^{\mathsf{PRF},0}_{\mathcal{A},F}(k) = 1 \mid \neg E\right] \cdot \Pr[\neg E].$$

If the event E does not occur then the simulation provided to $\mathcal{A}$ is exactly the same as $\mathsf{Exec}^{\mathsf{PRF},1}_{\mathcal{A},F}(k)$ since the random function $\mathcal{R}$ and the black box queries to F_{s^*} are answered in exactly the same way (since H is a random oracle). Hence $\Pr[\mathsf{Exec}^{\mathsf{PRF},0}_{\mathcal{A},F}(k) = 1 \mid \neg E] = \Pr[\mathsf{Exec}^{\mathsf{PRF},1}_{\mathcal{A},F}(k) = 1]$. We then bound the terms $\Pr[\neg E] \le 1$ and $\Pr[\mathsf{Exec}^{\mathsf{PRF},0}_{\mathcal{A},F}(k) = 1 \mid E] \le 1$. Finally $\Pr[E] = q_H/2^k$ where q_H is the number of random oracle queries made by $\mathcal{A}$ and $q_H \le \tau$ gives

$$\Pr\left[\mathsf{Exec}^{\mathsf{PRF},0}_{\mathcal{A}_\tau,F}(k) = 1\right] \le \Pr\left[\mathsf{Exec}^{\mathsf{PRF},1}_{\mathcal{A}_\tau,F}(k) = 1\right] + \frac{\tau}{2^k}$$

giving the desired result that

$$\nu_k(\tau) \le \frac{\tau}{2^k}.$$

To show $\varepsilon_Q(\tau) = \tau/2^m + \tau/(2^{m-Q})$ we consider an adversary who is given some value y and Q bits of a preimage x. The adversary has access to the random oracle G used to instantiate φ_Q. There are two ways such an adversary can compute a value $\tilde{x}$ such that $G(\tilde{x}) = y$; he could query G on x or he could query G on some other value x' that is also a preimage of y. Querying G on x happens with probability at most $\tau/2^{m-Q}$, where τ is the maximum number that can be made to G, since the adversary would have to find the remaining value of the preimage from a space of size 2^{m-Q}. Querying G on some other preimage x' happens with probability at most $\tau/2^m$. Combining these gives the desired result that

$$\varepsilon_Q(\tau) \leq \frac{\tau}{2^m} + \frac{\tau}{2^{m-Q}}.$$