Foundations, Properties, and Security Applications of Puzzles: A Survey

ISRA MOHAMED ALI, MAURANTONIO CAPROLU, and ROBERTO DI PIETRO, Hamad Bin Khalifa University (HBKU), College of Science and Engineering (CSE), Division of Information and Computing Technology (ICT)

Cryptographic algorithms have been used not only to create robust ciphertexts but also to generate cryptograms that, contrary to the classic goal of cryptography, are meant to be broken. These cryptograms, generally called puzzles, require the use of a certain amount of resources to be solved, hence introducing a cost that is often regarded as a time delay—though it could involve other metrics as well, such as bandwidth. These powerful features have made puzzles the core of many security protocols, acquiring increasing importance in the IT security landscape. The concept of a puzzle has subsequently been extended to other types of schemes that do not use cryptographic functions, such as CAPTCHAs, which are used to discriminate humans from machines. Overall, puzzles have experienced a renewed interest with the advent of Bitcoin, which uses a CPU-intensive puzzle as proof of work. In this article, we provide a comprehensive study of the most important puzzle construction schemes available in the literature, categorizing them according to several attributes, such as resource type, verification type, and applications. We have redefined the term puzzle by collecting and integrating the scattered notions used in different works, to cover all the existing applications. Moreover, we provide an overview of the possible applications, identifying key requirements and different design approaches. Finally, we highlight the features and limitations of each approach, providing a useful guide for the future development of new puzzle schemes.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Security and privacy** → **Cryptography**; **Security protocols**;

Additional Key Words and Phrases: Cryptographic puzzle, CPU-bound puzzle, Memory-bound puzzle, Network-bound puzzle, Proof-of-Work, Bandwidth-bound puzzle, Human-bound puzzle, CAPTCHA, Bitcoin, Cryptocurrency

ACM Reference format:
Isra Mohamed Ali, Maurantonio Caprolu, and Roberto Di Pietro. 2020. Foundations, Properties, and Security Applications of Puzzles: A Survey. *ACM Comput. Surv.* 53, 4, Article 72 (August 2020), 38 pages.
https://doi.org/10.1145/3396374

This publication was partially supported by Award No. NPRP 11S-0109-180242, from the QNRF-Qatar National Research Fund, a member of The Qatar Foundation. The information and views set out in this publication are those of the authors and do not necessarily reflect the official opinion of the QNRF.

Authors' address: I. M. Ali, M. Caprolu, and R. Di Pietro, Hamad Bin Khalifa University (HBKU), College of Science and Engineering (CSE), Division of Information and Computing Technology (ICT), Qatar Foundation, Education City, Doha, P.O. Box 5825, Doha, Qatar; emails: {isali, mcaprolu, rdipietro}@hbku.edu.qa.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

0360-0300/2020/08-ART72 $15.00
https://doi.org/10.1145/3396374

1 INTRODUCTION

The concept of "*puzzle*" was introduced in the field of security by Merkle in 1978, when he proposed a puzzle to reach key agreement over insecure channels [88]. Since the mid-'90s, puzzles have witnessed a growing interest by the research community in a variety of security fields ranging from cryptography and network security to computer performance and bio-metric technologies.

Puzzles constitute the core of many security protocols. They have been proposed as a security tool to achieve various goals, including defending against large-scale attacks, delaying the disclosure of information, creating uncheatable benchmarks, achieving consensus, and differentiating between humans and internet bots.

A puzzle is a moderately hard problem that is much easier to verify than to solve. Two key features of puzzles are the asymmetry and adjustability of workload. The solver is forced to dedicate a non-trivial amount of resources to find a solution, while the verifier can check its validity and correctness with a much-reduced effort. The adjustability of its hardness enables the verifier to tune the minimal amount of time and resources to be spent by the solver.

The process of spending resources to solve a puzzle generally introduces a time delay and an economic cost. The combination of these two effects makes puzzles a powerful tool that can be utilized to limit the capability of an adversary and prevent him from gaining significant influence. The first to observe this phenomenon were Dwork and Naor [42] in 1992, who proposed a type of puzzles, called *pricing functions*, as a solution to combat spam. In a similar context, *client-puzzles* were later proposed by Juels and Brainard [63] to mitigate denial of service attacks. The general idea of such puzzles is to associate a cost for each resource allocation request by requiring the client to complete a task before the server performs any expensive operation, hence making large-scale attacks infeasible.

One of the areas where puzzles have the most impact is in cryptocurrencies and other emerging technologies, such as blockchain. The idiosyncratic features of the puzzles, known as *proofs-of-work*, have paved the way to the implementation of a fully decentralized peer-to-peer cryptocurrency system. The idea of using puzzles in the creation of a digital-cash payment system has been long investigated [34, 114, 120], but only successfully implemented with the start of the Bitcoin project [96] in 2008. The puzzle is used to secure the public ledger of transactions by requiring miners to find a solution before being able to add a block to the ledger. The computational cost imposed by the puzzle prevents a computationally bounded adversary from double-spending transactions or effectively rewriting the ledger. The uniqueness and scarcity provided by the puzzle gives the currency an economic value and enables the process of minting currency [98].

Another common adaptation of puzzles is in Bot detection by web-based services. Companies such as Google, Yahoo, and Paypal, use a special type of puzzles, widely known as CAPTCHAs, to verify that the user is not a computer program. This type of puzzles enables human identification by using AI-hard problems that, ideally, cannot be solved by machines but can be easily solved by a simple human interaction [126]. The human feature provided by this type of puzzles is leveraged to slow down attackers and to prevent abuse caused by malicious bot programs masquerading as humans.

The early puzzle designing approaches concentrated on computational problems that are evaluated by the number of CPU-cycles required to find a solution. An example of such puzzles is the one used in many proposals, including Bitcoin, and was initially introduced by Back [10] in the Hashcash system in 1997. The puzzle requires finding an input to a hash function that produces an output with a specific number of leading zeros. A major drawback of such puzzles is the possible mismatch in the level of processing speeds over time and between different types of processors [41]. This problem was addressed by Abadi et al. [1] in 2003, who introduced an alternative computational approach that relies on memory-latency, known as *memory-bound functions*. Since memory-latency values are normally more stable than CPU-speeds, most recent systems will solve the puzzle at a similar speed. Another approach was to rely on network latency by Abliz and Tznati [2]. Subsequently, several works that rely on memory and bandwidth were presented in different research areas.

Although there have been several proposed construction schemes and a wide-range of applications for puzzles, to the best of our knowledge there has not been any attempt to characteristically distinguish puzzles from other related notions. In this article, we define the term "puzzle" as an umbrella name subsuming all moderately hard functions that are relatively easier to verify than to solve.

Contributions. This work can be seen as an extensive introduction to puzzles, providing the reader with a theoretical background and an overview on the different types of puzzles. Our work aims to fill the gap between the growing works on puzzles and the lack of a comprehensive survey that covers the different types of puzzles. Another aim is to clarify the connections and differences between the terms and notions used to describe a puzzle. We summarize our contributions as follows:

- We collect and integrate the scattered notions of puzzles into a uniform introductory work.
- We determine the criteria for puzzle categorization.
- We provide an overview of the applications of puzzles and identify the key requirements and challenges faced in each application field.
- We examine the different approaches used in the design of puzzles and identify the features and limitations of each one, ideally inspiring the development of novel, effective puzzle schemes.

Roadmap. The rest of this article is organized as follows. Section 2 provides an introductory overview on puzzles and describes their different types. Section 3 lists the properties and idiosyncratic features of puzzles. Section 4 provides a survey on the applications of puzzles, states the key requirements for each field, and discusses the viability of puzzles in each application field. Section 5 provides an in-depth survey of the state-of-the-art construction schemes and further developments. Finally, Section 7 concludes the survey by summarizing our contributions.

2 FOUNDATIONS AND BACKGROUND

Unlike traditional security problems and algorithms, such as cryptograms, which ideally cannot be cryptanalyzed, a puzzle is defined as a problem that is meant to be solved [88]. It is easy to verify, such that it is easy to determine if the given inputs produce the given outputs, but moderately hard to solve, such that it is solvable in a reasonable time [42]. Solving it involves performing a number of operations that require a specific amount of resources, such as CPU cycles, memory, bandwidth, and human's attention. The terms "easy," "moderate," and "hard" are relatively used, since their exact definitions depend on the application and implementation of the puzzle.

The solution of a puzzle *in some cases* serves as a proof of work (PoW), in which it demonstrates to one party (the verifier) that the other party (the prover) has performed a specific amount of computational work in a pre-defined time interval [61]. In this sense, the purpose of puzzles differs from the standard cryptographic objective of showing the possession of a secret to proving the ability to expend a certain amount of resources within a certain time interval instead [61]. Anyone without the secret can solve the puzzle but only in one way, which includes dedicating a minimal amount of resources. In the following, we describe the abstract structure of puzzles and present their different categories.

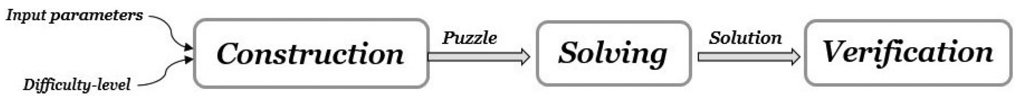

Fig. 1. Phases of a puzzle scheme.

2.1 Abstract Structure of Puzzles

The abstract structure of a puzzle scheme involves two parties, a **verifier** and a **prover.** The verifier determines the parameters of a puzzle and checks the correctness and validity of the solution submitted by the prover. The prover solves the puzzle to prove that it is a legitimate party or to obtain a specific reward.

A puzzle consists of three main components: **input parameters**, a **difficulty-level parameter**, and a **tunable function.** Input parameters are application-related data, such as the message in spam defense or block of transactions in cryptocurrencies. The difficulty-level parameter plays a role similar to that of a security parameter in a cryptosystem [42]. It determines the hardness of a puzzle and its effectiveness. The issuer should be able to tune the difficulty level flexibly according to the threat level and to accommodate Moore's Law, such that the tunable function adapts to the increasing amount of computational power and resources over time. In general, tuning the difficulty involves adjusting the size of the solution search space or revealing different degrees of the puzzle's solution.

Any puzzle scheme can be abstracted into three main phases, as illustrated in Figure 1: **Construction**, **Solving**, and **Verification.** In the Construction phase, a puzzle may be constructed by the verifier or the prover, depending on the scheme type (whether it is interactive or non-interactive). It may also include some offline pre-computations to reduce the online construction cost. This phase provides the two parties the information needed to execute subsequent phases. Once the puzzle is constructed, the prover starts performing the required operations to find a solution and then submits it to the verifier within the specified time interval. Finally, the verification phase involves verifying the validity and correctness of the submitted solution.

The execution of these phases is done through a protocol that can be either interactive or non-interactive. The former involves multiple rounds of communication executed by both parties; the prover and the verifier. It terminates by either accepting or rejecting the submitted solution, which is decided by the verifier in the verification phase. Non-interactive puzzles involve only one round of communication that is either initiated by the prover, which constructs and solves the puzzle then sends the solution to the verifier, or by the verifier, which constructs a puzzle that does not require an explicit verification and then sends it to the prover. The latter is a special type of non-interactive puzzles, known as implicit puzzles, where the verification is determined by the ability of the prover to perform a certain task that can be done in the absence of the verifier, such as in time-lock puzzles [115]. We discuss the verification type and the interactivity of puzzles in Section 2.2.3.

2.2 Types of Puzzles

Puzzles can be categorized based on their *application*, the *resources* required to solve them, or the *verification type* of the scheme. The application determines the way in which a puzzle can be utilized. It also defines the requirements and desirable properties of the puzzle scheme. The resources required to solve the puzzle define the metric by which its hardness is measured, whether it is computational steps, memory accesses, memory space or bandwidth, and so on. Finally, the verification type refers to the means by which the solution of a puzzle is used and whether it requires implicit or explicit verification. The rationale behind having more than one categorization is that none of the cited aspects is directly related to each other. The resource type is not directly determined by the application type, and vice versa. Furthermore, the verification type is determined by the application field and not by the application type itself. Therefore, providing a single categorization that combines any of these aspects may not be possible. In the following, we present the different types of puzzles with respect to the aforementioned aspects and discuss their characteristics as well as the relationships between the three aspects: application, resource type, and verification.

2.2.1 Application. Puzzles have been relied on by several security protocols in the literature. Historically, puzzles have been utilized as a *pricing tool* to assign a certain cost for accessing a resource or service, as a *delaying tool* to delay accessing a specific resource, as a *metering tool* to meter the access of a specific resource, as an *identity assignment tool* to achieve consensus in decentralized systems, and as a *human identification tool* to discriminate against humans from bots.

In the following, we list the different types of puzzles based on the way they are applied and present a brief description of each.

- ***Pricing puzzle:*** As defined by Dwork and Naor [42], it is a function that is moderately hard to compute and requires a known lower-bound expenditure of resources. It is not amenable to amortization and cannot be computed more efficiently after some pre-processing. The difficulty of computing a pricing puzzle is leveraged to increase the cost of launching automated large-scale attacks. It is usually applied in contexts where the low cost of using a service leads to abuse.
- ***Delaying puzzle:*** Also known as time-lock puzzle [115], it is a moderately hard function that requires a precise amount of time *(real time, not CPU-time)* to compute. It can only be computed sequentially by performing a deterministic number of computations, and cannot be solved significantly faster with large investments in hardware. The number of computational steps required to find the solution is predetermined by the puzzle issuer allowing him to control precisely when the prover can access the "locked" resource.
- ***Timing puzzle:*** As first introduced by Franklin and Malkhi [49], it is a moderately hard function that requires performing computations incrementally with increasingly large efforts invested. At every stage of the computation, a solver can generate a solution that verifies that a given state is certainly the current state of the computation [21]. The number of computation steps is determined in the solving phase instead of being fixed during the construction phase (as in delaying puzzles). The difficulty of solving a timing puzzle is used to ensure the security of a metering/measuring method.
- ***AI-hard puzzle:*** As described by Ahn et al. [126], it is a function that can generate and verify problems that a defined portion of the human population can solve but current computer programs cannot solve. Ideally, it is hard for a machine to compute, which allows ensuring that a human is in the communication channel and not a bot.

Puzzles provide several features that may be exploited to achieve a specific effect. For example, solving a puzzle requires spending resources hence introducing a time delay that allows the verifier to define when the prover may access the protected service. This feature is exploited by several schemes to achieve the timing effect. Generally, the application of a puzzle is motivated ***by one or more*** of the following features:

- ***Computation:*** This feature requires dedicating a defined amount of resources to solve the puzzle. It is utilized to associate a cost to a specific service or activity, such as sending an email or mining cryptocurrencies.

- ***Timing:*** This feature requires the prover to invest a specific amount of time in solving the puzzle. It is utilized to introduce a delay or to measure the time spent on a specific activity, such as visiting a website.
- ***Human:*** As the name implies, this feature requires human interaction to find a solution, since it cannot be solved automatically by machines. It is utilized to differentiate between humans and machines.

The timing feature is tightly related to the other features, however, schemes that are mainly motivated by time are not affected by the amount of resources used to solve the puzzle. In particular, puzzles that are used as a timing or a delaying function cannot be solved faster using more resources, such as multi-processors in parallel computing. However, pricing puzzles, that are used as a pricing function, are designed to ensure the prover performs a certain amount of computational work that introduces a cost and limits the rate of a specific attack to the amount of resources available to adversaries. Unlike timing puzzles, the time required to solve a pricing puzzle is probabilistic, where it has a predictable expected time but a random actual time [10]. Finally, human puzzles, such as CAPTCHAs [126], are the only type of puzzles that has the human feature, where they are mainly designed to ensure the prover is a human. We would like to note that human puzzles may be considered as computational puzzles, where the computational work is performed by the human and the dedicated resource is his attention.

2.2.2 Resource Type. Puzzles are either *CPU*, *memory*, *bandwidth*, *network*, or *human* bound (AI hard).

- **CPU-bound puzzles:** the computational work is quantified by the number of CPU cycles required to find the solution, which varies vastly in time according to Moore's law, as well as across different machines.
- **Memory-bound puzzles:** the computation is evaluated either by the number of memory accesses or the amount of memory space required to solve a puzzle. Therefore, the computation speed of these schemes is bound by memory latency and bandwidth.
- **Bandwidth-bound puzzles:** are evaluated by the amount of bandwidth dedicated to solve the puzzle.
- **Network-bound puzzles:** the time required to solve the puzzle is bounded by network latency as the solving process involves sending and receiving packets in a certain order.
- **Human-bound puzzles:** are puzzles that cannot be computed by artificial intelligence but can be easily solved by a simple human interaction. They are evaluated not by the amount of computational resources but by the amount of attention a human dedicates to solve the puzzle.

In the general case, the resource type is neither directly related to, nor determined by, the selected application type, since different types of resources can be used in the same application field. For example, there exist several implementations and proposals of puzzles in decentralized cryptocurrencies that are bounded by different resources including CPU [10], memory [15], and human interaction [18]. However, for specific applications, such as Bot detection, the only type of resource that can be utilized to discriminate between humans and machines is human interaction. Furthermore, the same type of resource can be used in different applications, such as CPU, which bounds pricing puzzles applied in DoS defense, delaying puzzles applied in time-release cryptography, and timing puzzles applied in uncheatable benchmarks.

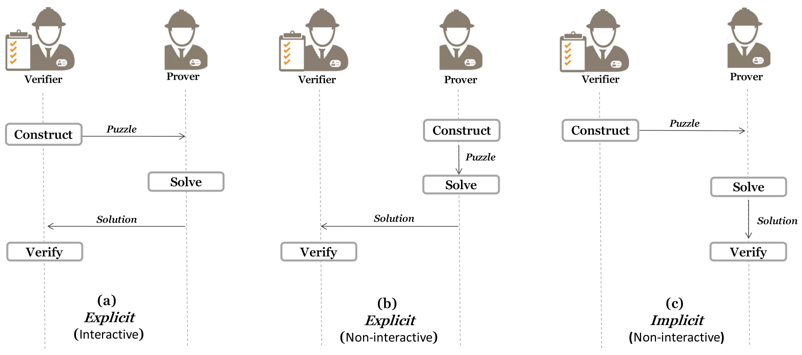

Fig. 2. Types of puzzle schemes based on the verification and interactivity: (a) is an explicit interactive scheme, (b) is an explicit non-interactive puzzle scheme, and (c) is an implicit non-interactive puzzle scheme.

2.2.3 Verification Type. The verification of a puzzle scheme can either be *explicit*, where it is performed by the verifier or *implicit*, where it is determined by the ability of the prover to successfully complete a specific task without the involvement of the puzzle issuer [61].

In Figure 2, we illustrate the different types of puzzle schemes based on the verification type and interactivity. An explicit puzzle scheme can be executed through both interactive and non-interactive protocols, while an implicit puzzle scheme is only executed through a non-interactive protocol, since it requires one round of communication only. Consider **(P, V)** as a two-party protocol through which a puzzle scheme is executed, where **P** is the prover and **V** is the verifier. An interactive puzzle scheme is considered as a two-message[1] **(P, V)** protocol, while a non-interactive puzzle scheme is considered as a one-message **(P, V)** protocol.

In an *explicit* two-message **(P, V)** protocol, **V** executes **`Construct`** and then sends the generated puzzle **`Puzz(`**I_n**`,t, k)`** to **P**, where I_n is a set of n application related input parameters, t is the maximum time required to solve the puzzle, and k is the difficulty level parameter. Upon receiving the puzzle, **P** executes **`Solve`** and then sends the produced solution S to **V.** The protocol terminates by **V** executing **`Verify(S,`** I_n**`, t, k)`** that outputs either `accept` or `reject`.

In an *explicit* one-message **(P, V)** protocol, **P** executes both **`Construct`** and **`Solve`**, then sends a message containing both the generated **`Puzz(`**I_n**`,t, k)`** and the produced solution S**.** Upon receiving the message, **V** executes **`Verify(S,`** I_n**`, t, k)`** that outputs either `accept` or `reject`.

Finally, in an *implicit* one-message **(P, V)** protocol, **V** executes **`Construct`** and then sends the generated puzzle **`Puzz(`**I_n**`,t, k)`** to **P. P** then executes both **`Solve`** and **`Verify(S,`** I_n**`, t, k)`**, where the latter outputs either `success` or `failure`.

In the following, we discuss each verification type and provide some examples.

- **Explicit:** In an explicit puzzle scheme, the solution serves as a way of convincing the verifier that a specific amount of effort is spent. The scheme can either be *interactive* of multiple communication rounds or *non-interactive* of one communication round, where the prover generates and solves the puzzle in the absence of the verifier. A non-interactive puzzle can be applied in an interactive scheme as a challenge-response protocol, while the converse is not possible. In non-interactive schemes, it is necessary to ensure that the prover cannot effectively control the puzzle generation nor he can precompute the solution. This can be achieved by referencing the puzzle generation to a public source of randomness. An example

[1]We highlight that such a protocol involves at least two messages, which may increment based on the designed scheme and application field.

of an explicit puzzle scheme that can be used in both interactive and non-interactive settings is the Hashcash proof-of-work scheme [10], which is also used in Bitcoin [96].

- **Implicit:** In an implicit puzzle scheme, anyone, including the prover, can verify a correct solution without the participation of the puzzle issuer [61]. The verification is determined by the prover's ability to carry out a particular task, such as decrypting a message using the key obtained from solving time-lock puzzles [115]. An implicit puzzle scheme is considered as a special type of non-interactive puzzle schemes, since it requires only one communication round.

The type of verification is determined by the specified application field. In particular, explicit verification is required when the solution is used as a proof to *convince* the verifier that the prover did solve the puzzle. This includes pricing puzzles used for DoS defense and cryptocurrencies, AI-hard puzzles used for Bot detection, timing puzzles used to produce uncheatable benchmarks, and delaying puzzles used to produce timestamps. Contrarily, the delaying puzzle used in time-release cryptography does not require explicit verification, since the solution reveals a key that is used to decrypt a ciphertext. The solution does not serve as a proof of work, but rather as a way to guarantee that the key is only obtained after a certain amount of computational time has passed.

3 IDIOSYNCRATIC FEATURES

The fundamental requirement of a puzzle, as first introduced, is that it should be easy to verify but moderately hard to solve. With the evolution of puzzles in various security fields, many properties and desirable requirements were defined to form an effective puzzle. In the following, we list these properties and provide a brief description of each one. The first seven properties are the fundamental properties that define a puzzle, that are common among all types of puzzles, while the rest are essential for specific application fields, but not for all applications.

(1) ***Asymmetry:*** This property describes the nature of a puzzle, as introduced by Dwork and Naor [42]. The amount of work required by each party is asymmetric; it should be much easier for the verifier to produce a puzzle and verify a solution than for the prover to find a solution.
(2) ***Granularity and parameterization [42]:*** There should be proper parameters that could be adjusted to allow the puzzle to scale with Moore's law. The verifier should be able to flexibly adjust the puzzle difficulty according to the threat level against the underlying protected service. For instance, if the difference between two adjacent difficulty levels is large, then increasing the difficulty would add a huge workload on legitimate parties. Linear-grained puzzles have the highest density of difficulty levels, while exponential-grained puzzles have the lowest density.
(3) ***Amortization-Freeness [42]:*** The prover should not be able to produce multiple solutions with a cost equivalent to that of producing one solution.
(4) ***Independence [42]/Correlation-free [51]:*** Solving a puzzle or knowing the solution of previous puzzles does not help in solving others.
(5) ***Efficiency:*** This describes the efficiency of a puzzle measured in terms of cost and overhead introduced to both parties of the puzzle scheme, as follows:
    (a) ***Low construction and verification cost:*** It is not enough to maintain a work gap between the verifier and prover but also to not incur a burden on the verifier that subjects it to resource-exhaustion attacks.
    (b) ***Stateless [63]:*** A stateless puzzle scheme does not require the verifier to store any information to be able to verify a solution, which is desirable in constrained environments.

(c) ***Memory-less [17]:*** A memory-less puzzle scheme does not require the verifier to access the main memory for verification. This property is desirable for memory-bound/hard schemes to enable fast verification and allow low-memory verifiers to participate in the scheme.
(d) ***Minimum interference [51]:*** The operations performed to solve a puzzle should not interfere with concurrently running applications.
(e) ***Minimum communication complexity [121]:*** Refers to the bandwidth and number of rounds required to exchange puzzles and solutions between the two parties. A puzzle scheme should have very low communication complexity and should not add significant traffic to the network.

(6) ***Unforgeability [32]:*** Initially introduced by Chen [32], the prover should be unable to forge a puzzle in a way that allows him to precompute the solution. The lack of this property makes the puzzle ineffective in many applications, such as mitigating DoS attacks.
(7) ***Freshness [48]:*** This property is also referred to as tamper-resistance by Feng et al. [48], which indicates that the puzzle's solution is not valid indefinitely and cannot be reused by other provers. In other words, the puzzle scheme should be resilient to replay and precomputation attacks.
(8) ***Uniqueness [48]:*** Having a unique solution for each puzzle is essential in contexts such as time-release crypto, since the solution is used as a key to decrypt the ciphertext.
(9) ***Puzzle fairness [2]:*** As defined by Abliz and Znati [2], the time required to solve a puzzle should be similar for all solvers despite their available resources (CPU, memory and bandwidth). This property eliminates the disparity problem between a powerful attacker and a legitimate prover.
(10) ***Non-parallelizability [115]:*** The puzzle can only be solved in sequential steps and cannot be solved in parallel using multiple machines. This property is necessary for achieving the timing effect and ensuring puzzle fairness. It ensures the effectiveness of a puzzle against high-end adversaries, who utilize parallel computing to solve the puzzle faster.
(11) ***Deterministic [28] (Low-cost variance):*** For delaying and timing puzzle schemes, such as time-lock puzzles [115], the number of operations performed to solve a puzzle should be deterministic to control the amount of computing time required. Other types of puzzles may have a probabilistic cost, where solving the puzzle has a predictable expected time but a random actual time. In general, lower variance in cost provides better control over the puzzle difficulty and assures puzzle fairness.
(12) ***Progress-free [17]:*** In applications, such as cryptocurrencies, the scheme is required to have a random probability distribution, such that the probability of finding a solution is independent of the amount of effort already spent in solving the puzzle.
(13) ***Interactiveness [61]:*** A puzzle can be either interactive or non-interactive. The former requires the verifier to send the challenge (such as client puzzles against DoS), while the latter can be constructed without the need for the verifier's participation (such as PoWs used in blockchain systems).
(14) ***Publicly Verifiable [61]:*** In some non-interactive puzzle schemes, such as Hashcash, the verification can be performed by any other party without the participation of the puzzle issuer. This property is essential in decentralized systems, where any participant in the network can verify the solution without requiring a central authority or server.
(15) ***Trapdoor-based [42]:*** A trapdoor mechanism is used in some puzzles to lower the complexity of puzzle verification by storing secret information that reduces the amount of time needed in solving the puzzle significantly. This property may preclude public auditing, where the verifier has a conflict of interests, such as in web visit metering [49]. A

Table 1. Categorization of Puzzles

Categorized by	***Application***	***Resource Type***	***Verification***
Types	• Pricing function • Delaying function • Timing function • AI-hard function	• CPU-bound • Memory-bound • Bandwidth-bound • Network-bound • Human-bound	• Explicit • Implicit

Table 2. Key Requirements of Puzzles for Each Application Field

	Key Requirements								
Application Field	*Easy to construct*	*Fine-grained*	*Deterministic computation*	*Non-parallelizable*	*Non-interactive*	*Publicly verifiable*	*State -less*	*Trapdoor -less*	*Fair*
Spam defense	✓				✓				
DoS defense	✓	✓					✓		✓
Cryptocurrencies	✓				✓	✓	✓	✓	
Delayed Disclosure		✓	✓	✓					
Auditable metering	✓					✓		✓	
Uncheatable Benchmarks			✓	✓				✓	
Bot Detection	✓								✓

trapdoor-free puzzle provides trust in decentralized systems, such as cryptocurrencies, by ensuring that the issuer has no advantage in solving the puzzle and cannot forge the proof.

4 APPLICATIONS

The idea of using moderately hard problems that are much easier to verify than to solve have been long investigated in various application fields. Puzzles are referred to, in the literature, as proofs-of-work (PoWs), timing functions, delaying functions, cost/pricing functions, AI hard functions and CAPTCHAs. Each term is used to describe the application of a puzzle. Nevertheless, they all share the same property of being moderately hard to solve that a polynomial-time party is capable of finding a solution by dedicating a specific amount of resources. Most puzzles were first designed for a specific application, however, researchers are currently investigating the possibility of designing a multipurpose moderately hard function that can be applied in several security fields [4].

In this section, we review the significant applications of puzzles and categorize them based on the way a puzzle is used *(as a pricing, delaying, timing or AI function)*, as shown in Table 1. We also discuss the key requirements and the viability of each type in each application field. Given the findings of the study we have conducted on puzzles and their utilization in a wide range of application fields, we conclude that all types of puzzles should be asymmetric, parameterizable, amortization and correlation free, unforgeable, fresh, and efficient.[2] For each application, there are specific requirements and properties that are not as important as in other fields. In addition, the level of efficiency in terms of construction cost and the granularity of the puzzle are relative to the application field, where the puzzle's effectiveness in some applications is restricted by the easiness of its construction, by the ability to finely tune its difficulty, or both. In Table 2, we present the key requirements for each application field.[3]

[2]Please refer to Section 3 for the description of each of the listed properties and requirements.

[3]We highlight that Table 2 is derived based on the study we have conducted on the application aspect of puzzles, which includes a review and an analysis of the different construction schemes targeting the specified application fields, in addition to, works that study the challenges faced by the employment of puzzles as a security mechanism in some application fields.

4.1 Pricing Puzzles

The idea of pricing puzzles is to impose a cost for accessing services that can be easily abused by attackers. The requester of a service is charged with the amount of resources required to find the solution of a problem that is much harder to solve than to verify. In what follows, we present the different security fields that pricing puzzles are applied in and discuss the key requirements and viability of each.

4.1.1 Key Agreement. The notion of *"puzzle"* was first introduced by Merkle [88], in 1978, as a method for key agreement over insecure channels. The objective is to allow any two parties to agree on a secret key that will not be known to eavesdroppers. In this scenario, the protocol initiator plays the role of the verifier, in which he constructs a puzzle consisting of N encrypted keys and verifies the solution submitted by the other party (prover). The other party solves the puzzle by selecting one of the encrypted keys and decrypting it using brute-force, then sending its ID to the verifier. Without knowing which key is mapped to that ID, an eavesdropper must decrypt the N keys at random until he encounters the correct one, which requires an effort of $O(N^2)$. Unlike other puzzle schemes, Merkle's puzzle is required to be infeasible for any polynomial-time party. This requirement makes the method insecure, however, it offers the feature of workload adjustability. The puzzle issuer can control the solution cost by adjusting the difficulty parameter. This feature is exploited by several works to provide a light-weight pairwise key agreement protocol for resource-constrained environments, such as wireless sensor networks [109, 134] and low-energy Bluetooth devices [105].

4.1.2 Spam Defense. In 1992, Dwork and Naor [42] suggested using puzzles as an access control mechanism. They introduced the concept of *pricing functions* that increase the cost of sending emails to mitigate spam. Their approach is fundamentally an economic one, in which the processing time dedicated to solve the puzzle is a finite resource. Therefore, spammers are limited to the amount of computing resources they can afford, which prevents them from sending emails in bulk. It is a non-interactive explicit puzzle scheme, where the prover is the sender of an email and the verifier is the recipient. The sender is required to construct the puzzle using time, destination, and message as input parameters, then find a solution and send it along with the email. The recipient verifies the attached solution and accepts the email only if the solution is valid.

In 1997, Back [10] rediscovered the idea and implemented the Hashcash system, that was originally intended for spam and DoS defense, and currently used in Bitcoin [96]. Hashcash is a non-interactive trapdoor-free proof-of-work scheme that has an unbounded probabilistic solution cost. Requiring a puzzle for each email would not only disincentivize spammers but also prevent legitimate mass mailing, since it requires a significant expenditure of resources. To solve this problem, Dwork and Naor [42] introduced the idea of a trapdoor-based puzzle, which is much easier to compute given some secret information. Legitimate bulk mail can only be sent cheaply by a centralized trusted authority that holds the secret information. While Hashcash is a decentralized solution that provides better efficiency and resiliency against pre-computation attacks, it does not solve the legitimate mass mailing problem. Nevertheless, it has been deployed in several projects including SpamAssasin[4] and Penny Post.[5]

Key Requirements. To be applied as an anti-spam mechanism, a puzzle scheme should not incur a significant burden on legitimate parties. It should be efficient and non-interactive to avoid requiring the recipient to interact with the sender before receiving an email. It should also consider the sharp disparities across computer systems, as observed by Abadi et al. [1], that could

[4]https://spamassassin.apache.org.
[5]http://pennypost.sourceforge.net/PennyPost.

make the scheme ineffective against powerful spammers with powerful hardware, restrictively slow for legitimate clients with regular personal computers. They suggest using memory-bound puzzles, because memory access speeds differ significantly less than processor speeds. Further developments on memory-bound puzzles in spam defense are proposed in References [41, 43] and discussed in Section 5.2.

Viability. Laurie and Clayton [71] analyzed the viability of pricing puzzles as an anti-spam mechanism and concluded that a universal scheme where every email carries a fixed-cost puzzle is impractical for two reasons. First, a significant proportion of legitimate users are also affected by the added cost preventing them from accessing the service. Second, malicious users can steal CPU cycles by accessing insecure machines, using someone else's resources to solve puzzles. Instead, they suggest incorporating puzzles with other techniques, such as whitelists, to vary the hardness of the puzzle and reduce the burden on legitimate senders. They also suggest using human-bound puzzles, such as CAPTCHAs [126], which are presumably more difficult to steal. However, several research papers have demonstrated the feasibility of pricing puzzles against spam when used in parallel with other techniques, such as reputation systems [79].

4.1.3 DoS Defense. Client puzzles are a type of pricing puzzles used to defend against denial of service (DoS) attacks. Namely, resource depletion attacks that prevent the victim from processing legitimate requests for a service in a server-client setting. Before allocating any resources for a given request, the server requires the client to commit a portion of its resources by solving a puzzle. Requests that do not include a correct solution are dropped. Legitimate clients may experience a degradation in service, however, attackers are unable to send a large number of requests simultaneously due to the time delay introduced by the puzzle.

The two key features of a puzzle that qualify it as an anti-DoS mechanism is workload adjustability and asymmetry. The former allows the server to tune the difficulty level of the puzzle according to the current threat level, by initially setting it to zero when there are no attacks and increasing it as the intensity of the attacks increases. While the latter shifts the workload from the server to the client as, it is much easier for the server (verifier) to construct and verify a puzzle than for the client (prover) to solve the puzzle.

The first construction scheme of client puzzles was proposed by Jules and Brainard [63] as a countermeasure to connection depletion attacks. Aura et al. [9] later improved and generalized the design of the puzzle to employ it in any authentication protocol. Many construction schemes have been subsequently proposed to protect various types of services including network IP and TCP channels [48, 87, 99, 131, 132], TLS [37, 100], capability-granting channels [102], and key agreement protocols [118]. Despite the various techniques used by these construction schemes, they all must satisfy the fundamental properties described by Feng et al. [48].

Key Requirements. For a puzzle to be applied in DoS defense, it should be efficient enough to guarantee the availability of the puzzle distribution service and avoid subjecting the scheme itself to a DoS attack. It should be resilient to precomputation attacks, where the puzzle solution indicates that the computational effort was recently spent. Furthermore, robust authentication mechanisms must be employed to prevent attackers from spoofing puzzles and disabling the server by falsely triggering the puzzle mechanism against it [47].

The granularity of the puzzle, that represents the density of difficulty-levels, should be high to allow the server to finely control the amount of computational effort spent by the client. Finally, forcing all clients to solve puzzles before allowing access is crucial to mitigate the attack, however, not all clients have the same capabilities. Therefore, the scheme must also consider low-power clients and adjust the level of puzzle complexity to match the client's capabilities to achieve puzzle fairness [47].

Viability. The main challenge faced in deploying puzzles as a DoS countermeasure is determining and setting the appropriate difficulty level that limits the abilities of attackers but not legitimate parties. From an economic perspective, the effectiveness of a pricing puzzle can only be achieved when the amount of work required from legitimate clients and attackers differ significantly [71]. The construction scheme must identify and discriminate against known malicious behavior [47]. Most client puzzle schemes set the difficulty based on a single metric, such as the load on the system [37, 63, 102], the rate at which the client sends requests [48, 64], or the level of demand for the service [131, 132]. However, using a single metric has been proven to be insufficient [54], as it provides clients weak access guarantees at high per-request overhead.

The viability of client puzzles remains an open question. Issues such as constructing puzzles efficiently while ensuring non-parallelizability, adjusting the puzzle difficulty to prevent subversion while maximizing server utilization, and attaining equitable fairness must all be addressed to incentivize its deployment.

4.1.4 P2P Systems and Cryptocurrencies. In the context of decentralized systems, pricing puzzles are used to address various security issues including Sybil [40] and collusion [110] attacks, achieving consensus in a Byzantine setting, and incentivizing correct behaviour by requiring participants to submit a puzzle solution that serves as a *proof-of-work* and then rewarding them for participation. This type of puzzles is utilized by important applications in creating decentralized cryptocurrencies, such as the recent systems Bitcoin [96], Ethereum [135], and Litecoin [74], or prior schemes such as the Micromint system of Rivest and Shamir [114].

Decentralized dynamic systems are highly vulnerable to the Sybil attack [40], whereby the attacker exploits the low cost of forging multiple identities that allows him to control a substantial fraction of the system and execute further attacks to subvert the system. Pricing puzzles have been long investigated as a decentralized Sybil defense mechanism in a variety of p2p network settings and overlays, including structured [7, 23, 76, 116] and unstructured [96, 135] overlays. The idea is to impose a computational cost on maintaining an identity within the system, hence limiting the proportion of Sybil nodes to the proportion of resources that an adversary can control per time unit.

In p2p identity systems, such as SybilControl [76], the puzzle is used as a distributed admission control mechanism that grants nodes the permission to join and stay functional in the system. In this protocol, all nodes are required to periodically solve a unique puzzle and collectively verify solutions of other nodes. If a node fails to compute the puzzle within the specified time interval, then its identity gets revoked. Decentralized cryptocurrencies, such as Bitcoin [96] and Ethereum [135], utilize pricing puzzles to achieve several goals at once. They exploit the scarcity and uniqueness provided by pricing puzzles to create economic value and mint crypto-currency. More importantly, they use the pricing puzzle as a key component in the blockchain protocol to achieve consensus and prevent double spending [98].

Blockchain is described as a cryptographic data-structure in which a transaction ledger, that is shared and agreed on by all nodes of the network, is recorded. Compared with the original design of identity pricing puzzle schemes [7, 23], the puzzle in a blockchain network is not used in the identity verification of participating peers. Instead, the peers are expected to collectively verify puzzle solutions broadcasted by other peers to determine who's block will be considered as the next block in the chain. The Sybil and double-spending attacks are mitigated by associating a computational cost to the process of adding a block to the chain. A comprehensive survey of the consensus protocol design and the effects of blockchain networks is presented by Wang et al. [130].

The cryptocurrency is created by nodes, known as *"miners,"* through a process that involves processing a block of transactions and finding a solution to a pricing puzzle that makes the block valid. Once a miner solves the puzzle, it broadcasts the proposed block to all other nodes, and receives a reward only if the block is accepted by the majority of nodes. The validation of that block is done independently by each node in the network, which includes verifying the puzzle solution and correctness of each transaction within the block. Miners are incentivized to act honestly, since incorrect puzzle solutions would result in the rejection of the block by the majority of nodes, hence losing the reward and wasting the effort spent in solving the puzzle. Further technical details on Bitcoin and digital currencies is presented in Reference [124].

The rate at which blocks are appended to the ledger is determined by the difficulty of the puzzle. It is adjusted such that, it is hard enough for an adversary to interfere and alter the system, but easy enough for miners to construct new blocks and unify their views of the public ledger. The robustness of the consensus protocol relies on the assumption that more than 51% of the computational resources are possessed by honest participants who follow the longest chain rule. Formal analysis of blockchain's security under different network assumptions appear in References [52, 53, 92, 103].

Key Requirements. Efficiency is one of the key requirements of a puzzle scheme to be applied in dynamic decentralized P2P systems. The puzzle should be easy to construct and verify, stateless, and compact to provide scalability for the underlying protocol. Given the lack of a trusted third party in such environments, the puzzle should be *non-interactive*, *trapdoor-free*, and *publicly verifiable*. Any node in the network should be able to efficiently verify the puzzle solution of any other node without the access to a trapdoor or any secret information. The trapdoor-free property is essential to provide trust in the system.

The freshness property of the puzzle should be ensured at the execution phase [130]. In particular, the puzzle solution should be non-reusable and unpredictable such that, the computational work is guaranteed and the proof is unforgeable. Furthermore, fairness should be ensured, such that the probability of finding a solution is directly proportional to the computational power of the node at any given time. This is crucial for cryptocurrencies, since solving the puzzle has an economical value. Finally, the hardness of the puzzle should be adjustable to adapt to the changing scale and settings of the p2p network.

Viability. Notably, the most impactful application of pricing puzzles to this date is in the implementation of permissionless p2p systems, namely, blockchain emerging technologies. Although examples such as Bitcoin and Ethereum demonstrate the success of pricing puzzles in practice, several concerns are raised by the research community regarding the stability of these systems [22], the high power consumption [83], and wastage of resources that increases proportionally to the system's popularity [46]. Furthermore, the parallelizability nature of the utilized puzzle schemes allows nodes to increase their voting power by using customized hardware (such as ASICs) that solves the puzzle substantially faster. This development in hardware subverts the pricing puzzle approach and implies several threats. In particular, it diminishes the democratic value of decentralized cryptocurrencies by suppressing low-end nodes [124] and enables powerful nodes to collude and alter the system [45]. Several puzzle schemes have been proposed to address these issues including, ASIC-resistant puzzles [17, 113], non-outsourceable puzzles [91], useful puzzles [12, 12, 90], and eco-friendly puzzles [18, 44].

4.2 Delaying and Timing Puzzles

The timing feature of a puzzle is exploited by several schemes either to slow down attackers, to lock resources for a precise amount of time, or to measure the time spent accessing a resource. Unlike pricing schemes, delaying and timing schemes are concerned with making *CPU time* and *real time* agree with an approximate precision. They achieve this by using inherently sequential problems that have a deterministic solution cost. It is important to note that the solution time of these puzzles is *approximately controllable*, since different computer systems operate at different speeds. In the following, we discuss the different application fields of these types of puzzles.

4.2.1 Uncheatable Benchmarks and Auditable Metering. The first proposal in designing a puzzle that requires solving an inherently sequential problem seems to appear in the application of hardware benchmarking in 1993 by Cai et al. [28, 29]. The idea is to validate the performance of specific hardware by having it compute a puzzle that reflects its computation power. The asymmetry feature provided by puzzles allows customers with low-end machines to verify the benchmark of high-end computer vendors. A customer provides the hardware vendor with a puzzle, who finds a solution and submits it as a verification of the claimed hardware performance. The customer then verifies the puzzle and checks that the solution time is indeed within the claimed bound. The soundness of the puzzle scheme would guarantee that the vendor could not cheat by optimizing his code or modifying it.

Franklin and Malkhi [49] proposed the idea of metering client accesses via a timing puzzle. They presented a lightweight solution to the problem of forged client website visits. The timing puzzle requires an incremental amount of computations, which makes forging a large number of client visits expensive and time-consuming. The solution of the puzzle serves as evidence that a specific amount of time has passed. At each webpage visit, the client is asked to compute a timing function that requires performing repeated hashing incrementally until the end of the visit. The result is then sent to an auditing proxy that verifies its correctness. The main drawback of their construction scheme is the requirement of reconstructing the puzzle for accurate verification, which was later addressed by Chen and Mao [31].

Key Requirements. In addition to being non-parallelizable and deterministic, a timing puzzle must provide the ability of applying it incrementally to reflect the actual continuous-time being spent. The solver should be able to produce extendable solutions, where the puzzle difficulty is not set in advance but is incremental with the time spent in computing it. Forging access duration or performance records should require a known amount of resources that increases proportionally to the amount of forgery. For public auditing, the puzzle should be trapdoor free, where there is no shortcut available for the prover to find a solution with fewer resources.

Viability. Timing puzzles may be considered as an unreliable metering method due to the existence of several uncontrollable factors that may affect the accuracy of real-time measurement, including network delay, bandwidth, and computational power [20]. The existing timing puzzle schemes only offer lightweight security requiring precise limits on the adversary's processing speed.

4.2.2 Time-Release Cryptography and Time-stamps. The idea of time-release cryptography is to "send data into the future" by encrypting a message that can only be decrypted after a predefined amount of time has passed. In 1996, Rivest et al. [115] implemented this idea using time-lock puzzles, which can be applied to delay sealed-bid auctions, digital cash payments, and key escrow. They have also been proposed for other applications such as, enabling offline submission [62], providing pseudonymous secure computation [67], constructing a two-round concurrent non-malleable commitment [78], and supporting digital forgetting [5].

Time-lock puzzle is a cryptographic primitive that allows locking data, making it accessible only after a certain delay. It is an implicit trapdoor-based puzzle scheme, where the solution reveals the key that decrypts the encrypted data. In Reference [115], the puzzle issuer selects the desired time delay and constructs a modular exponentiation puzzle that can only be solved by performing t modular squaring operations sequentially. The number of squaring operations can be exactly controlled hence providing the ability to finely tune the difficulty of the puzzle. The verification is not explicitly required, however, it can be done more efficiently by anyone who can access the trapdoor that is created in the construction phase. Inspired by time-lock puzzles, Mahmoudy et al. [85] introduced the concept of *proof of sequential work* (PoSW), an explicit time-lock puzzle scheme, which enables the solver to prove that a specific number of computation steps was performed sequentially on a given challenge. The solution of the puzzle is publicly verifiable and indicates that an approximate number of time units have passed since receiving the puzzle. They propose using PoSW to produce *relative* timestamps for documents, whereby a timestamp d of a document at time T proves the existence of that document at time $T - d$. A stamper specifies the desirable duration d, constructs and solves the puzzle by using the document as an input parameter. The verifier then checks the validity of the timestamp by verifying the solution of the puzzle.

Further developments on such construction schemes have been proposed to provide better efficiency [33] and uniqueness [106]. Uniqueness is important to guarantee that the solver cannot produce multiple solutions at the same cost of one. Boneh et al. [21] study the problem of constructing delaying puzzles, which they refer to as *"verifiable delaying functions,"* and present further applications of such puzzles, including randomness beacons, proof of replication, and resource-efficient blockchains.

Key Requirements. The key requirements for such applications are *non-parallelizability* and hardware-independence. The solution time should not depend on the amount of hardware being used. A solver who uses a large amount of investments in hardware, namely, parallel computing, should not be able to find a solution substantially faster than the pre-determined time. Finally, the time required to construct and verify the puzzle should be much less than the solution time.

Viability. The main challenge that may hinder the deployment of a delaying puzzle in a specific application is the requirement of exact guarantees on the precision timing, which cannot be achieved due to the existence of variations in the speed of single computers. For other settings where there are no trusted third parties available, the puzzle construction schemes proposed in Reference [115] are unsuitable, since generating the puzzle requires knowing the (*secret*) puzzle solution in advance and verification requires accessing a trapdoor. Although the schemes proposed in [33, 85] provide public verifiability, their security is only proved in the random oracle model and the uniqueness of the produced solution is not guaranteed.

4.3 AI-hard Puzzles

AI-hard puzzles, widely known as CAPTCHAs, are intended to ensure the presence of a human in a communication channel by using hard AI problems that differentiate between humans and bots. They are puzzles that are easy-to-solve for humans but hard-to-solve for automated computer programs. In the following, we present the main security fields in which human-bound puzzles are used and discuss their key requirements.

4.3.1 Bot Detection in Web-based Services. The first to suggest using a reverse Turing test for verifying that a human is the one requesting a service over the web was Naor [97] in 1996. Several practical examples were then proposed and developed [11, 77, 107]. In 2000, Von Ahn et al. [19] introduced the notion of a CAPTCHA and provided a formal framework that models it as a hard AI problem in Reference [126]. CAPTCHA can be considered as a trapdoor-based explicit puzzle scheme whose difficulty is based on an AI problem that can only be solved by a human. A typical CAPTCHA puzzle is constructed by first generating a random target solution, such as a text or an image, and then performing distortion techniques to that solution to make it hard for computers to solve the puzzle. The target solution, that is generated in the construction phase, is later used by the verifier to check the correctness of the solution submitted by the prover.

CAPTCHAs are used by Google and many other popular web-based services to mitigate abuse caused by malicious bot programs masquerading as humans. Such abuse includes automated account registration, password guessing attacks, systematical database mining, and massive voting in polls. Before being able to access the service, a client is required to prove that he is a human by solving a given instance of the puzzle. The human feature provided by this type of puzzles slows attackers down and prevents them from sending a large number of fake requests generated by automated programs.

Many variations of construction schemes exist in the literature that are based on different AI problems, including reading distorted text [127], audio recognition [72, 89], human face recognition [55], and emergent-image recognition [50]. A comprehensive review of the different categories and sub-applications of CAPTCHA is presented by Hidalgo and Alvarez [58]. Despite the many variations of AI-hard puzzles, such schemes are based on the hypothesis that the underlying AI problem cannot be solved by the adversary's machine more accurately than what is currently known in the AI field [126].

Key Requirements. To provide both practicality and usability, the puzzle should be easy-to-construct for the puzzle generator machine and easy-to-solve for the human solver. Humans should be able to solve the puzzle effortlessly with a negligible error rate. The hardness of the puzzle should be adjustable such that both robustness and usability are guaranteed even with the advance of technology. The best-known programs for solving the underlying AI hard problem should fail on a non-negligible portion of the puzzles, despite that the method of constructing the puzzle instances is known [97]. Finally, the puzzle scheme should be fair that does not discriminate against disabled people and involve different sensory abilities, including hearing and vision.

Viability. The main issue that faces the deployment of AI-hard puzzles in web-based services is setting the appropriate difficulty that ensures both resistance against machine-learning attacks and human usability. The same distortion methods used to make the puzzle unsolvable by machines can also significantly degrade human usability [138]. CAPTCHAs are often hard for humans to solve due to a number of demographic factors such as age, language, and education [27]. Furthermore, this type of puzzles fails to recognize people with visual and hearing impairments as humans, which prevents them from accessing the underlying protected web service [58]. These usability issues may drive customers to abandon services that deploy CAPTCHAs resulting in financial losses for those companies.

Although many construction schemes have been successfully attacked using machine-learning techniques [26, 93, 136, 137], a significant gap between human intelligence and the current artificial intelligence still exists. However, this does not ensure the security and effectiveness of the puzzle scheme as it is still vulnerable to human relay attacks, whereby the puzzle is outsourced to paid human-solvers [35]. Motoyama et al. [95] analyzed the behavior and dynamics of CAPTCHAs from an economic perspective. They conclude that CAPTCHA is a low-impact mechanism that reduces the attacker's profitability hence minimizing the cost and legitimate user impact of more expensive secondary defenses. However, they may be ineffective in scenarios where the profit gained from launching the attack is much greater than the cost associated with paying humans to solve the puzzle.

4.3.2 *Decentralized Cryptocurrencies.*

Recently, Blocki and Zhou [18] introduced the concept of *proof-of-human-work* (PoH), a human-in-the-loop puzzle that is publicly verifiable and can be used to build a decentralized cryptocurrency system. It is a non-interactive explicit puzzle scheme that can only be solved with sufficient human assistance. Unlike the traditional standalone CAPTCHA, the solution of the puzzle is unknown to the puzzle-generator machine and the difficulty is adjustable.

The scheme involves two types of puzzles, a CAPTCHA and a pricing puzzle similar to that used in Bitcoin. To produce a valid block, the miner is required to first prove that he is a human by solving a CAPTCHA instance, then use the obtained solution as an input parameter to the pricing puzzle. The CAPTCHA instance is constructed obliviously using the *universal samplers* developed by Hofheinz et al. [59]. The instance is generated along with a verification tag, and the corresponding solution remains concealed in the obfuscated program. This feature provides public verifiability, where anyone in the network can verify the submitted solution. The difficulty of the puzzle is adjusted by having the verifier reject a valid solution with a certain probability so that the human miner have to generate and solve a specific number of puzzles to produce a valid proof-of-human-work.

The human feature provided by the AI-hard puzzle is exploited to satisfy properties that traditional cryptocurrencies such as Bitcoin lack, which are eco-friendliness, usefulness, and centralization-resistance. Eco-friendliness and usefulness are achieved by relying on human effort instead of computational power. The human effort may involve performing educational [68] or productive [60] tasks to avoid wasting human cycles. While centralization-resistance is achieved by ensuring fairness, where any two humans are capable of performing a similar amount of work to solve the puzzle.

Key Requirements. In the context of cryptocurrencies, the AI-hard puzzle should be efficient and trapdoor-free, such that it is easy for a machine to construct, but difficult for any machine (*including the puzzle-generator machine*) to solve without sufficient human assistance. It must be non-interactive and publicly verifiable such that a machine can easily verify the puzzle solution and ensure that the issuer does not already have the solution without any human assistance. Finally, the puzzle scheme should provide a sufficient density of difficulty-levels to allow adjusting its hardness according to the changing scale and settings of the system.

Viability. The viability of AI-hard puzzles in decentralized cryptocurrencies is affected by two main challenges. First, a trapdoor-free construction scheme whereby the solution is unknown to any party throughout the generation of the puzzle is required to provide trust in the system. This challenge is addressed by the authors of HumanCoin system [18] using indistinguishably obfuscation (IO); however, the current development achievements in IO do not provide a practical solution, hence their approach is currently impractical. Furthermore, their proposed system requires an initial trusted setup phase, since the party generating the system may embed a trapdoor allowing it to produce coins without involving any human work. Second, the security and stability of the system rely on the hardness of the underlying AI problem. The life of such cryptocurrency is anticipated to be shorter than other cryptocurrencies that depend on cryptographic primitives, since AI breakthroughs are achieved more frequently. Therefore, achieving and maintaining trust using AI-hard puzzles in such cryptocurrency systems remains an open question.

5 CONSTRUCTION SCHEMES

As discussed previously, puzzles may be categorized based on several aspects including the ways they are applied, the type of verification, and the type of resource that bounds the scheme. In this section, we categorize them based on the resource that bounds the puzzle scheme, which includes CPU, memory, and bandwidth. We survey state-of-the-art puzzles by focusing on the type of construction scheme and present further developments and improvements on these schemes.

5.1 CPU-Bound Schemes

5.1.1 Merkle Puzzles. Computational puzzles were first introduced by Merkle [88] as a method to establish a secure key agreement over insecure channels. The main designing goal was to create a work gap between legitimate parties and passive adversaries to guarantee secure key distribution. The puzzle is based on symmetric cryptography and is constructed by generating N encrypted keys, each having a unique ID. The other legitimate party solves the puzzle by selecting one of the keys and decrypting it by brute-force then submitting its ID. To know the key, an eavesdropper must decrypt all N keys, since he cannot determine which of the IDs is mapped to the key. The puzzle's difficulty is adjusted by adjusting the size of the key used to perform the encryption. As a key agreement protocol, this scheme is impractical and insecure, since the optimum work gap that can be achieved using this method, as proven by Barak and Mahmoudy [13], is quadratic. However, it was the building block towards public key cryptography. Merkle puzzles are different than the following puzzles as both parties are required to perform a similar amount of work $O(N)$, hence it is not asymmetric.

5.1.2 Pricing Functions. Dwork and Naor [42] were the first to suggest using puzzles as an access control mechanism to combat email spam. They used the puzzle as a "pricing function" to assign a computational cost to resource allocation requests. Their goal is to increase the cost of sending an email for spammers by forcing them to compute a unique puzzle for each recipient. They proposed three puzzle schemes. The first scheme is based on factoring a square root and the other two are based on digital signature schemes with smaller security parameters. The first is verified by squaring the submitted solution, while the digital-signature-based schemes are verified using trapdoors to reduce the verification cost. The availability of trapdoors in their scheme is essential, not only to reduce the verification cost but to allow legitimate mass mailing by specific authorities. Solving the puzzle requires forging a signature without actually breaking a private key using a moderately hard algorithm such as, the Pollard algorithm. The proposed schemes satisfy the amortization-freeness property, but are prone to pre-computation attacks, which weakens the effectiveness of the puzzle. Furthermore, similar to previous schemes, it suffers from inefficiency requiring relatively high construction cost.

5.1.3 Time-lock and Delaying Functions. Rivest et al. [115] rediscovered the idea of puzzles to implement time-release cryptography and introduced the notion of *time-lock puzzles*, which are computational puzzles that can only be solved in a precise amount of time. Their designing goal was to create a puzzle that can only be solved sequentially by performing a deterministic number of operations. The puzzle is used to allow encrypting a message that can only be decrypted (unlocked) after a predetermined period of time specified by the puzzle issuer elapses. This type of puzzles is different than traditional PoW schemes, in which the solution of the puzzle serves as a decryption key rather than a method of convincing the verifier. In particular, the verification is not performed by the verifier but determined by the ability of the prover to decrypt the encrypted message using the solution of the puzzle.

The hardness of Time-lock puzzles relies upon the infeasibility of factoring large integers. To construct the puzzle, the issuer first sets its parameters by generating a composite modulus N as the result of multiplying two large private prime numbers p and q. It determines the time required to solve the puzzle $t = TS$, where S is the number of squaring operations current machines can run per second, and T is the desired time specified by the puzzle creator. The issuer then generates the puzzle by encrypting a message m with key K and encrypting the key as $C_k = K + a^{2^t} mod\ N$. The puzzle is given as (N, a, t, C_k, C_m) and the solution of the puzzle reveals the key that decrypts the message. Solving it requires performing t modular squaring sequentially starting with a. The difficulty of the puzzle is adjusted by increasing or decreasing t, which provides fine granularity. Knowing $\Phi(N)$, the verifier can use the trapdoor given by Euler's function to compute the solution in $O(\log N)$ modular multiplications.

Time-lock puzzles guarantee non-parallelizability and deterministic computation, which provides fine-grained difficulty adjustment. However, the requirement of generating large prime numbers and performing modular exponentiations for every puzzle is resource-exhausting, which makes it impractical and unsuitable for resource-constrained environments.

Efficiency. To reduce the construction and verification cost, Karame and Capkun [66] proposed using an RSA key pair with a smaller exponent and a semi-prime modulus. They based their puzzle

$$h(C, N_S, N_C, X) = \overbrace{000\ldots000}^{\text{the } k \text{ first bits of the hash}} \underbrace{Y}_{\text{the rest of the hash bits}}$$

Fig. 3. Hashcash puzzle construction [61].

construction on the intractability assumption in RSA, which states that computing a private exponent when the semi-prime modulus is less by multiple orders of magnitude than the public key is computationally infeasible. Given k as a security parameter, the verification cost in the proposed scheme is reduced by a factor of $\frac{|N|}{k}$. Despite the significant cost reduction, it is still sufficiently expensive that it cannot be deployed in large-scale environments. Furthermore, their scheme does not provide fine-grained control over the difficulty level as, the gap between two subsequent difficulty levels is significantly increased compared to the previous time-lock puzzle scheme [115]. Further efficiency improvements on modular-exponentiation-based puzzles are proposed in Reference [108]. Tang and Jackmans [121] introduced different verification modes for the construction phase proposed in Reference [115] to increase the verification efficiency and make it suitable for a server-client communication model.

5.1.4 Hashcash and Client Puzzles. Inspired by the idea of pricing functions [42], Juels and Brainard [63] proposed client puzzles to mitigate connection depletion attacks. The objective is to have the clients commit their resources before establishing a connection, by requiring them to solve computational puzzles for each request. Since the adversary would send a great number of connection requests, it must solve each puzzle associated with each request. Therefore, precluding attackers from overwhelming the server and allowing legitimate clients to establish a connection. Unlike previous puzzles, client puzzles do not incur high cost on the verifier, in which they are easy to construct and can be constructed in a stateless way.

The server constructs the puzzle using a hash function that processes a bitstring X of length l as input and produces a hash image Y, illustrated in Figure 3. The puzzle is given as $(X < k+1, l >, Y)$, where the first k bits of X are hidden. Clients must perform a brute-force search for the k missing bits that produces Y and submit the solution with the request. The problem of solution precomputation is addressed by embedding a time-stamp in the puzzle and requiring the client to submit the solution within a specified time interval. The difficulty is tuned by increasing/decreasing the number of bits to be searched for. To decrease the probability of guessing the solution and to have a finer difficulty adjustment, they suggest composing the puzzle of m independent subpuzzles, each requiring a unique k-bit solution. This composition may increase the difficulty for an attacker in guessing the solution, but the granularity is still coarse having the difficulty level grow exponentially. Furthermore, the efficiency of the puzzle decreases as the number of sub-puzzles increases.

Efficiency. Aura et al. [61] improved the client puzzle's efficiency by reducing its length and minimizing the number of hash calls needed for both construction and verification. They generalized the design of the puzzle to employ it in any authentication protocol. The puzzle is generated by the client, unlike the previous scheme [63], where the server generates both hash input and output. In this scheme, the client is given a nonce N_s and is required to find a bit-string X that if hashed together with the nonce (and other client data) produces a hash image with k leading zero bits. For verification, the server performs a single hash call to check if the submitted solution produces k leading zero bits. These improvements motivated further practical implementations of client puzzles. Dean and Stubblefield [37] provided a compatible implementation of this scheme to protect TLS servers, Moskowitz et al. [94] adapted it in the Host Identity Protocol (HIP), and Wang and Reiter [132] integrated it in the network layer to mitigate bandwidth-exhaustion attacks. One additional feature that this construction offers, as proposed by Back [10], is that it can also be used in a non-interactive setting, where the prover chooses the challenge and the solution can be publicly verified.

Despite the efficiency improvements, this puzzle construction, which was initially implemented by Back in the Hashcash system [10] and currently used in Bitcoin [96], provides coarser difficulty levels disabling the verifier from flexibly adjusting the difficulty according to the threat level. Furthermore, it is highly parallelizable and has a probabilistic solving cost with high variance. These drawbacks may prevent puzzle fairness and reduce the effectiveness of the puzzle. Parallelizability gives adversaries, with higher CPU-speed, a great advantage over legitimate parties, while probabilistic cost does not guarantee that the time required to solve the puzzle is similar to all clients. The search process for n-bits might abort after the first try or after performing all of the 2^n tries.

5.1.5 Client Puzzle Variants. Further efficiency improvements were proposed and the aforementioned problems of client puzzles were addressed by several researches. In this subsection, we categorize them based on the problems and puzzle features being discussed in each.

Determining the difficulty level. The difficulty of a puzzle determines its security and effectiveness. Setting the difficulty to a low level may reduce the workload on the adversaries, which allows them to successfully launch an attack, while setting it to a high level may deter honest parties from participating in the underlying application. Given the existence of an adversary with unknown computing power, it may be difficult to properly adjust the difficulty level without increasing the cost for legitimate clients. Wang and Reiter [131] addressed this problem by using an auction mechanism. They introduced the concept of "*puzzle auctions*," where each client is allowed to set the hardness of the puzzle it solves. The server then assigns its resources to the client that solved the hardest puzzle (the one with the highest difficulty level). They assume that a legitimate client would expend more resources than a compromised zombie client, since attackers will not increase the workload of these zombies above a certain threshold to avoid detection. They argue that this mechanism is effective, where clients can win an auction by raising their bids just above the attacker's bid. However, it gives an unfair advantage to clients with higher computation power and, potentially, to powerful attackers. In addition, it adds more rounds to the scheme by requiring the client to submit more than one puzzle solution of increasing difficulty levels until a connection is established.

Granularity. Feng et al. [48] addressed the coarseness problem of hash-based puzzles by providing the client with a hint along with the puzzle. The hint is a value near to the solution, which reduces the brute-force search space. To solve a puzzle, the client starts at the hint and searches the range linearly for the solution. Puzzle difficulty can be adjusted finely by adjusting the accuracy of the hint. This scheme provides better control over the difficulty level, but it can only be applied interactively requiring the verifier to generate a puzzle for each client instead of just sending a challenge as in Reference [61].

Reducing construction cost. To eliminate the computational load of generating puzzles from the verifier, Waters et al. [133] proposed puzzle outsourcing mechanisms. Their goal is to protect the puzzle scheme itself from being subject to a DoS attack and reduce client delay added by the previous puzzle schemes. The puzzle, which is based on the Diffie-Hellman problem, is constructed by a third party called bastion. The construction requires modular exponentiations while the verification, which is performed by the server, requires a memory lookup and one modular exponentiation. The client is required to invert a discrete logarithm using some partial information. This information is represented as a specific range of seed values. The client performs a brute-force search for the solution seed within the specified range. Since the verifier can control the size of the range, it can linearly adjust the difficulty of the puzzle. The client delay is reduced by allowing it to compute solutions offline and requiring it to solve a puzzle per time interval instead of per request. The per time interval requirement prevents adversaries from precomputing solutions. Although the construction is outsourced and the same bastion can generate puzzles that can be utilized by multiple servers, it is still expensive due to the modular exponentiation, making it inefficient and unscalable. In addition, the solution finding process in this scheme is similar to that in hash-based schemes, it requires performing an exhaustive search within a specific range, which is a highly parallelizable task.

Gao [51] suggested adding a pre-construction stage to allow the server to compute expensive operations in idle time and reuse the calculated parameters during online puzzle construction by combining them with time parameters. The author developed two trapdoor-based puzzle schemes, one is RSA-based and the other is based on the discrete logarithm problem. Both schemes require modular arithmetic calculations not only in the pre-construction stage but also in the online construction phase. As the previous scheme [133], they provide fast verification via a memory look-up and linear granularity, however, the solving process is also highly parallelizable.

Non-parallelizability. Generally, non-parallelizability is important to control the timing feature of a puzzle, where the solution time should be approximately controllable by the verifier. This property is crucial to allow the verifier to appropriately set the difficulty, which is not possible due to the huge disparity in computing power between machines. Non-parallelizability is important in applications that require puzzle fairness and prevent an adversary that uses parallel computing from solving puzzles significantly faster than the expected time. Non-parallelizable CPU-bound puzzle schemes are either based on an inherently sequential problem [115, 122] or utilize a chaining technique [56, 84].

Time-lock puzzles [115] are based on repeated squaring that ensures non-parallelizability, however, they cannot be employed in large-scale and resource-constrained environments due to their high construction cost. Two hash-based schemes that offer non-parallelizability were proposed by Ma [84] and Gorza and Petrica [56]. These schemes utilize puzzle chaining technique, which requires sequential solving steps to complete the whole puzzle chain. Although they were designed to reward the client for partially solving a chain, we also discuss the possibility of using this type of construction as one puzzle that has one solution. A puzzle chain consists of a set of puzzles that can only be solved in a specific order, in which solving a puzzle requires computing the solution of the predecessor puzzle. In each puzzle chain, there is at least one puzzle that does not depend on any other puzzle.

Ma [84] utilizes hash chain, which was initially introduced by Lamport [70], to construct the puzzle. The goal is to provide a receiver the control over which packets to receive. This is achieved by forcing the sender to invert a hash chain, where each inverted hash value permits sending m packets only. As a stand-alone puzzle, the construction cost is relatively high and the verification may require reconstructing the puzzle chain in some settings to avoid memory exhaustion attacks. In addition, adjusting the difficulty is not as flexible as the author claims, where inverting a chain of hash values may not always be achieved in reasonable time and setting the digest's size to lower number of bits (*16*-bits as suggested) lowers the difficulty level, which makes the process of finding a solution as easy as constructing the puzzle [122].

In Groza and Petrica's scheme [56], the puzzle chain is given as $([P_0, r_0], [P_1, r_1] \ldots, [P_n, r_n])$, where P is the puzzle and r is a string of random bits. The chain is constructed by first concatenating two state-dependent random values, ρ and r, then double hashing them to form the first puzzle in the chain $P_0 = H^2(\rho_0 || r_0)$. The remainder of the chain is created by XORing the result of hashing the previous state-dependent values ρ_{i-1} and r_{i-1} with two new state-dependent values ρ_i and r_i, then double hashing the result, $P_i = H^2((\rho_i || r_i) \oplus H(\rho_{i-1} ||| r_{i-1}))$. Solving the puzzle requires solving the chain in the order it was constructed, starting with P_0 then finding the ρ_i of each

P_i. Similar to the previous scheme, it is an inefficient stand-alone puzzle requiring the verifier to perform three hash operations for each puzzle in the chain for both construction and verification. Both schemes only guarantee partial non-parallelizability as, each puzzle in the chain can be solved in parallel, hence they do not actually solve the CPU-speed disparity problem.

To achieve non-parallelizability with cheaper construction and verification cost, Tritilanunt et al. [122] proposed a new puzzle scheme that relies on the subset sum problem, which is a variant of the knapsack problem. Given a set of objects, each has a specific weight, the prover must determine the number of objects that can be included in a fixed-size knapsack. In other words, given a set of positive integers $(a_0, a_1, \ldots, a_n)$ and a positive integer$S$, find a subset of a that sums up to S. Solving the puzzle can be done by bruteforce, which is highly parallelizable, however, the puzzle difficulty is set such that it is more efficient to apply Lenstra's lattice reduction algorithm *LLL* [75] instead. The algorithm requires recursive computations, hence it cannot be parallelized. Both construction and verification are cheap requiring one hash operation and some additions. This scheme provides non-parallelizability, deterministic computation and polynomial granularity. However, the *LLL* algorithm suffers from huge memory requirements that limit its deployment in general applications. In addition, it does not solve the resource disparity problem as, according to the authors, the solving algorithm may not be executed using platforms of a lower power than PCs.

5.1.6 Summary of CPU-bound Puzzles. Most CPU-bound puzzle schemes are based on cryptographic hash inversions [9, 10, 48, 63, 131] or digital signature algorithms [42, 51, 133]. In general, hash-based puzzle schemes are more efficient where the generation and verification requires an insignificant number of cryptographic hash operations. However, they suffer from three fundamental drawbacks. The first is coarse-grained difficulty adjustment, where adjacent difficulties vary by a factor of two. This may not allow the verifier to flexibly adjust the difficulty according to the threat level, hence reducing the effectiveness of the puzzle. Although some variants, such as Hint-based [48], provide linear granularity, they can only be applied interactively. The second is parallelizability, which introduces the CPU-speed disparity problem and gives powerful adversaries advantage over honest parties. Finally, the solution-finding process of these puzzles is probabilistic and its cost variance is very high that it does not guarantee fairness as some provers may get lucky and find the solution much faster than others. However, number-theoretic puzzles, such as time-lock puzzles [115], provide finer granularity and low solution cost variance, but are less efficient requiring the verifier to compute large integer modular exponentiations, which is unsuitable for resource-constrained environments. Considering parallelizability, both time-lock puzzles and knapsack-based puzzles ensure non-parallelizability; however, as discussed previously, the former requires the verifier to perform expensive operations, while that latter suffers from huge memory requirements for solving the puzzle.

The lack of an efficient and non-parallelizable CPU-bound function led researchers to investigate the utilization of other resources, such as memory [1, 41], bandwidth [2, 129], and human's attention [97, 126], which are discussed in the following sections. Table 3 summarizes the CPU-bound scheme categories and their properties.

5.2 Memory-bound and Memory-hard Schemes

Using memory-intensive computations in a puzzle scheme has been proposed by several works [1, 8, 44, 104], with different notions, to provide both equitable computation and resistance against specialized hardware, such as GPUs and ASICs. The solution cost of the puzzle is measured either by the number of accesses to the main memory or the amount of memory space rather than the number of computation operations.

Table 3. Summary of CPU-bound Schemes

		Properties								
Category	***Puzzle Scheme***	***Easy to construct***	***Easy to verify***	***Granularity***	**Low Cost Variance**	***Non-parallelizable***	***Non-interactive***	***Publicly verifiable***	***State -less***	***Based on***
Non-Hash Based	Merkle [88]	✗	✓	Exponential	✗	✗	✗	✗	✗	Symmetric Crypto.
	Pricing functions [42]	✗	✓	Exponential	✗	✗	✓	✗	✗	Digital Signatures
	Time-lock [115]	✗	✓	Linear	✓	✓	✗	✓	✗	Repeated Squaring
	Subset-Sum [122]	✓	✓	Polynomial	✓	✗	✗	✓	✓	Subset-Sum problem
Hash-Based	Client Puzzles [63]	✓	✓	Exponential	✗	✗	✗	✗	✓	Multiple Hash Inversion
	Hashcash [10]	✓	✓	Exponential	✗	✗	✓	✓	✓	Single Hash Inversion
	Hint-based [48]	✓	✓	Linear	✗	✗	✗	✗	✗	Single Hash Inversion
	Hash-chain [84][56]	✗	✓	Linear	✗	partial	✗	✗	✗	Hash chain reversal

Memory-bound puzzle schemes require a significant number of memory accesses to be solved. The solution time is bound by the memory latency, where the complexity is measured by the number of cache misses and not the actual amount of memory being employed. The adversary's goal in such a scheme is to decrease the number of memory accesses by either benefiting from cache or performing intensive computations instead. However, *memory-hard* puzzle schemes require a significant amount of memory space to solve. The complexity is measured by the number of memory locations being used for a given number of operations. The adversary's goal in such a scheme is to use less memory space by trading it for time or extra computations.

A memory-bound function may be considered as memory-hard in the sense that the amount of memory required is greater than the cache's size. While a memory-hard function is memory-bound only if the locality is good in its memory access pattern such that it results in a certain number of cache misses [113].

5.2.1 Easy-to-compute Functions. A well-known problem with CPU-bound puzzle schemes is the significant variations in the amount of computing power available to provers. Abadi et al. [1] addressed this problem and were the first to introduce the notion of memory-bound puzzles. The goal is to construct a puzzle that latest systems can compute at a similar speed by relying on memory latency instead of CPU-speed. They proposed a memory-bound puzzle that consists of an easy-to-compute pseudo-random function *F()*, which its inverse $F^{-1}()$ requires more time-consuming computations than accessing the memory (i.e., inverting it can be done more efficiently via the space-time tradeoff). The verifier selects an integer x_0 from the domain $[0 \ldots (2^n - 1)]$ and computes $x_{i+1} = F(x_i) \oplus i$, where $0 < i < k$. The puzzle is given as x_k and a checksum of the sequence $(x_0, x_1, \ldots, x_k)$. Solving the puzzle requires constructing a table for $F^{-1}()$ and working backwards from x_k to find a pre-image x_0', such that the checksum of the path from x_0' to x_k matches that of the challenge. Since there exist several pre-images that lead to multiple paths, the solver is forced to explore a tree of pre-images that has a depth k, root x_k and a total size of $O(k^2)$. The parameters n and k are chosen carefully to ensure that the table cannot be stored in cache, thus in the best case scenario, the number of cache misses required is also $O(k^2)$. Verification requires k forward computations of *F()*, which increases exponentially as the CPU-cost of *F()* decreases (i.e., the processing speed of the current machines increases according to Moore's Law). They also discuss a variant of this scheme that can be applied in a non-interactive setting. The challenge is not presented by the verifier, rather produced by a pseudo-random generator using some application related data, such as a message in combating spam, as the seed.

The main drawback of the scheme proposed in Reference [1] is the existence of a time-space tradeoff for inverting the function *F()*, which may allow adversaries to circumvent the scheme using higher computation power and only rarely accessing the memory. The puzzle complexity is highly affected by current processing speeds, as it requires more memory and higher verification cost to keep pace with Moore's law. Consequently, its efficiency decreases, which hinders its large-scale deployment and practical implementation. In addition, the work ratio between the two parties is quadratic and cannot be increased, since deeper trees allow the solver to benefit from cache and invert several values in the cost of just one memory access.

5.2.2 MBound. Dwork et al. [41] argued that easy-to-compute functions [1] may be solved with very few memory accesses that diminishes the memory latency effect and hence do not solve the problem. They further explored memory-bound functions and defined a class of functions based on *"pointer-chasing"* in a large random table T. The table T is of a fixed size that is approximately double the size of the largest current existing cache and is shared between the two parties. The solver is forced to perform random walks through T to find a path with specific characteristics. A walk is a path that requires making a series of sequential random accesses to T, where contents of the current accessed location determine the subsequent location to be accessed. Solving the puzzle is done by executing an algorithm (called *MBound*) that terminates successfully if after l walking steps the last n-bits of the hash output are zeros. The algorithm is executed k times until a successful path is found. The prover then submits the solution given as the hash-output along with the trial number k, which identifies the correct path. The verifier then explores the identified path k by performing l memory accesses to T and checks that the submitted hash value is correct and ends with n zeros. Using the random oracle model the authors prove a lower bound of $\Omega(2^n \cdot l)$ on the number of amortized memory accesses that an adversary must expend per puzzle. The number of walks to be performed by the solver is 2^n, thus the total expected cost for computing the puzzle is $2^n \cdot l$ cache misses, while the verification cost is l cache misses.

On a follow-up work, Dwork et al. [43] addressed the high communication complexity required for distributing and updating the incompressible large table by allowing the table to be constructed using graph pebbling. They observed that the table construction must also be memory-bound to prevent adversaries from exploiting the compact description of T and produce elements in cache whenever they are needed, only rarely accessing the memory. Pebbling is described as a game played on a directed acyclic graph (DAG). Finishing the game requires pebbling all output vertices (nodes with no children) of the graph.[6]

In this scheme, a pebble corresponds to a label of n-bits and placing a pebble corresponds to labeling a vertex by calling a hash function and storing the newly computed value in cache. Constructing the table requires pebbling the DAG D, where the labels of the output vertices are the elements of table T. The DAG has N input vertices (*numbered 1,2, …N*), N output vertices (*numbered* $N + 1, N + 2, \ldots, 2N$) and a constant indegree. The size of the graph is $O(n|T|log|T|)$ and consists of a stack of N-superconcentrators, which provide sharp tradeoffs having the time required to pebble the output vertices with less than N pebbles at least exponential in the depth of the graph. Therefore, the time required to pebble the outputs is superpolynomial in $|T|$ (the number of elements in T). Table construction is done by both parties only once, the table is then stored in main memory to execute the *MBound* algorithm [41]. This reduces communication cost, however, the requirement of constructing a table incurs a relatively high cost on the verifier and eliminates the easy-to-construct property.

[6]For more details on pebble games refer to Reference [81].

There are two main drawbacks of the aforementioned memory-bound puzzles that hinder their deployment. First, both verification and solving costs increase to accommodate to the current cache size, which reduces the puzzle's efficiency and disables legitimate parties with low memory resources from participating in the scheme. Second, the difficulty level cannot be adjusted flexibly, as it is constrained by the requirement of fine-tuning several parameters depending on the system's cache and memory configurations.

5.2.3 Pattern Database. Doshi et al. [39] noticed that the memory accesses described in prior memory-bound puzzle schemes are not associated with a known problem, hence they lack the algorithmic foundation that allows flexible tuning of difficulty parameters. They were the first to propose deriving memory-bound puzzles from heuristic search methods. The proposed construction scheme is based on a heuristic search technique using pattern databases for the Sliding Tile problem [82]. The Sliding Tile problem requires finding a path (using only the following moves: *left, right, up, down*) to slide the tiles on a grid from the initial state to the target state. Solving the problem can be done efficiently using existing computational algorithms, such as the A^* algorithm together with the Manhattan Distance heuristic. However, using a memory-based heuristic is more efficient, where the precise distance from the initial state to the abstract target state is computed and stored in a look-up table called *pattern database*.

Given a pool of target states ($G_0 \dots G_n$), the prover precomputes the pattern database that corresponds to these states offline. The verifier constructs the puzzle by randomly choosing f target states from the pool and performing d moves at random to each target G_i to create f initial states. The verifier then computes a message authentication code (MAC) over the performed moves and stores it in memory. The difficulty of the puzzle can be tuned by increasing or decreasing the number of target states f and the number of moves d. The puzzle is given as the f target states, the corresponding initial states, and checksums C_j, $1 \leq j \leq d$, where each checksum is computed over the jth move of each state. The prover then uses the computed pattern database to complete a guided search for every target state. All initial states are solved simultaneously as the prover must first deduce the right set of moves for a given difficulty level i, $1 \leq i \leq d$, such that the checksum of the corresponding moves matches C_j. Since different states are stored in different parts of the memory, forcing the prover to search for multiple states simultaneously results in cache misses hence ensures main-memory access. Once all the target states are reached, the prover submits the d moves performed on the f initial states. The verifier then checks that the moves are correct by computing the MAC of these moves and comparing it to the one stored.

Although the scheme provides better flexibility in tuning the puzzle difficulty, it is inefficient in terms of both construction and communication costs. For each puzzle, the verifier is required to perform $d * f$ moves and compute d checksums, while communication involves transmitting two sets of states (both initial and target) along with d checksums and a MAC.

5.2.4 Scrypt. In 2009, Percival [104] proposed Scrypt and introduced the concept of memory-hard functions that entail significant amount of memory to evaluate and require a large number of computations if less memory is utilized. Scrypt is a memory-hard key derivation function used for password hashing to increase the cost of brute-force dictionary attacks, whereby the attacker iterates through a number of likely passwords and apply the function to each password guess. The designing goal of the scheme is to reduce the advantage gained by adversaries who use custom-designed parallel circuits, while maintaining low per-evaluation cost of the honest user. This is achieved by requiring an amount of memory that is approximately proportional to the number of operational steps performed to evaluate the function.

The scheme consists of several memory-hard functions, we briefly describe the construction of ROMix, which constitutes the core of Scrypt. The basic idea of the algorithm, as described by

Percival, is to sequentially compute a large number of random values and then access each value randomly to ensure that they are all stored in RAM. Given a k-bit input value B, a chain of N input values $(X_0, \dots, X_{N-1})$ is computed as follows:

- $X_0 = B$ and $X_i = H(X_{i-1})$ for $i = 1, \dots, N-1$, where H is a hash function.

A chain of N output values $(V_0, \dots, V_n)$ is then computed as follows:

- $V_0 = H(X_{N-1})$, $V_i = H(V_{i-1} \oplus X_{V_{i-1} \ mod \ N})$ for $i = 1, \dots, N$, where V_N is the final output of the function.

The default strategy of computing Scrypt is to sequentially compute each X_i in the input chain and store it in the memory, and then compute each V_i of the output chain and fetch each X_i from memory as needed to produce the final output V_N. Two recent works show that Scrypt provides almost optimal resistance against ASICs from both the area [3] and energy [113] aspects. However, this is only achieved if the memory requirement of the scheme is large enough, which consequently incurs high construction and verification costs.

Scrypt is used as a pricing puzzle by several cryptocurrencies including Litecoin and Dogecoin, where the amount of memory is reduced to provide faster verification. To append a block to the chain, the miner is required to find a nonce n that if hashed along with the block's header B using Scrypt, produces a final output V_N that is less than a specific target value T. The solution is given as (B, n, V_N). Verification requires a single call to Scrypt and a single comparison operation to check the output value (V_N) against the target value T. The memory hardness of Scrypt can be set by increasing or decreasing the CPU/memory cost parameter N, while puzzle difficulty is adjusted by tuning the target value T.

Although a work gap between the two parties is maintained by having the prover perform multiple Scrypt calls to find the solution while only a single call is required for verification, the scheme is symmetric in terms of memory requirements. Furthermore, its adaptation in Litecoin cryptocurrency is not ASIC-resistant having ASIC mining rigs hundreds of times more efficient.

5.2.5 *Cuckoo Cycle.*

Tromp [123] proposed a memory-bound puzzle scheme based on finding constant sized cyclic subgraphs in a pseudo-random graph. The designing goal is to make mining cryptocurrencies on commodity hardware cost-effective by relying on memory-latency instead of computation speed in the evaluation of the puzzle. The puzzle is constructed by generating a directed bipartite graph with N vertices and M edges from a given set of input nonces using Cuckoo hashing [101]. The prover is required to find a set of L nonces whose corresponding edges form an L-cycle in the graph and whose hash digest is less than a given target hash value T, where $0 \leq T \leq 2^{256}$. The memory-hardness of the scheme is determined by the ratio M/N and the cycle length L, while the difficulty of the puzzle is adjusted by tuning the target hash value T. The verification cost is linear in L and independent of both M and N.

5.2.6 *Proof of Space.*

Dziembowski et al. [44] and Ateniese et al. [8] introduced the notion of *proof of space* (PoS) independently, each with a different definition and security guarantees. Generally, PoS is an interactive memory-hard puzzle scheme where the solution serves as a compact proof that the prover did possess and dedicate a significant amount of space. Unlike memory-bound schemes [1, 41, 123], PoS schemes rely on memory space instead of memory latency in the evaluation of the puzzle. Both PoS construction schemes are based on directed acyclic graphs (DAGs) that have high pebbling complexity and use the Merkle hash tree to enable efficient verification.

Given a challenge nonce *id*, the prover is required to build a graph G = (V, E) of N vertices and store the label of each vertex v computed as $l_v = H_{id}(v, \ H_{id}(l_{p1}, \dots, l_{pd}))$, where H_{id} is a hash function that depends on the identifier *id* and $l_{p1}, \dots, l_{pd}$ are the labels of v's predecessors (parents). The identifier is used to ensure that the same dedicated space cannot be leveraged for more than one proof. Once the labels are computed, the prover commits to these labels by constructing a Merkle hash tree $\tau = \tau^H(l_1, \dots l_N)$ and submitting the computed root of the tree, Φ, to the verifier. At this point, the verifier checks the consistency (correctness) of the root Φ by asking the prover to open the labels of $\boldsymbol{c}$ vertices and their predecessors. The opening of a label $l_{v_{c_i}}$ is the path from the root Φ to the leaf associated to vertex $\boldsymbol{v}$ in the Merkle tree τ. Given the labels of all the predecessors of vertex $\boldsymbol{v}$, the verifier can check if the label $l_{v_{c_i}}$ is correctly computed as described above. The solution is accepted only if all $\boldsymbol{c}$ openings and labels are computed correctly. To label a vertex v in graph G, the prover must compute and store the label values of all the predecessors $l_{p1}, \dots, l_{pd}$ of the vertex in memory. The labeling of the graph presents a proof that the prover has handled at least N space.

The definition of PoS introduced in Reference [44] extends that of Ateniese et al. [8] by including an additional phase, called *execution phase*, that allows the verifier to repeatedly challenge the prover and check if it is still dedicating the specified amount of space. The purpose of the additional phase is to allow honest provers to respond to the repeated challenges by accessing the N space, produced in the initialization phase, while using little computation hence addressing the high energy consumption drawback of CPU-bound puzzles. This formation of PoS is also referred to as proof of persistent space by Ren and Devadas [112].

Although these construction schemes provide strong security guarantees by forcing a cheating prover to either expend $O(N)$ space or time to produce an acceptable proof, they are both based on superconcentrators, which are relatively slow [16].

5.2.7 Equihash. Biryukov and Khovratovich [17] observed that any determined NP-complete problem can be used to design a memory-bound puzzle with tunable parameters, where its memory hardness is determined by the time-memory tradeoffs of the best known algorithms. They proposed Equihash, a proof-of-work scheme based on the k-dimensional generalized birthday problem [128]. In the generalized birthday problem, there are k sets of n-bit strings and the goal is to find k strings that XOR to zero. In Equihash, the k strings are generated randomly using the hash function H. The optimal solution algorithm to this problem has a time and space complexity of $O(2^{\frac{n}{k+1}})$ [128]), hence it is a memory-intensive algorithm. In addition, using $1/q$ less memory results in $O(q^{\frac{k}{2}})$ times more calls to the hash function, which limits the computation advantage of parallelization to the amount of memory bandwidth available. The verification requires performing 2^k hashs and XORs.

5.2.8 MTP-Argon2d. Biryukov et al. [16] proposed a *non-interactive* memory-hard puzzle scheme, called Merkle Tree Proof (MTP), based on the memory-hard function Argon2d [15]. They describe different instantiation settings of the scheme to be applied in crypto-currency, time-release cryptography and disk encryption. Similar to PoS schemes [8, 44], this scheme leverages a Merkle hash tree construction over an array of memory segments to allow fast and memory-less verification.

Given the challenge I, the prover constructs the puzzle by generating M segments of memory $B[1], B[2], \dots, B[M]$ using Argon2d from the given challenge. It then constructs a Merkle hash tree over the M segments committing to the segments' values. The root of the Merkle tree is declared as Φ. Each memory segment $B[j]$ is an output of of Argon2d's compression function F, which processes the preceding segment $B[j-1]$ and $B[\phi(j)]$, where $\phi(j)$ is a data-dependent indexing function that produces a memory segment index ranging in $[1 \dots j)$.

The prover solves the puzzle by accessing L memory segments of $B[1], B[2], \dots, B[M]$ pseudorandomly in the sequence $B[j_1], B[j_2], \dots, B[j_{L-1}], B[j_L]$ to find a nonce N that if hashed with

Table 4. Summary of Memory-bound Schemes

		Properties								
Category	***Puzzle Scheme***	***Easy to construct***	***Memory-less verification***	***Granularity***	**Low Cost Variance**	***Non-parallelizable***	***Non-interactive***	***Publicly verifiable***	***State-less***	***Based on***
Heuristic Search	Easy-to-compute functions [1]	✓	✓	polynomial	✗	partial	✗	✗	✗	Depth-first search
	Pattern Database [39]	✗	✓	polynomial	✗	partial	✗	✗	✓	Sliding Tile Problem
	Cuckoo Cycle [123]	✓	✓	exponential	✗	✗	✓	✓	✓	Graph Cycle Finding
	Equihash [17]	✓	✓	exponential	✗	partial	✓	✓	✓	k-XOR Birthday Problem
Graph Pebbling	Compact-MBound [43]	✗	✗	exponential	✗	partial	✓	✗	✗	Graph Pebbling
	Litecoin-(Scrypt)[74]	✓	✓	exponential	✗	✓	✓	✓	✓	Graph Pebbling Reduction
	MTP-Argon2d [16]	✓	✓	exponential	✗	partial	✓	✓	✓	Graph Pebbling Reduction
	Proof of space [44]	✗	✓	linear	✗	partial	✗	✗	✗	Graph Pebbling

the root Φ and the segments $B[j_i]$ produces a hash output Y_L with d trailing zeros (where j_i is determined from the nonce N, the root Φ, and the values of the preceding segments in the sequence). The solution is given as $(\Phi, N, \mathscr{L})$, where $\mathscr{L}$ is the opening of $2 * L$ memory segments $B[j_i - 1], B[\phi(j_i)]$. The verifier then validates the openings $\mathscr{L}$ and regenerates all $B[j_i] = F(B[j_i - 1], B[\phi(j_i)])$ to verify that Y_L has d trailing zeros. The memory hardness of the scheme is highly dependent on the hardness of Argon2d and can be adjusted by tuning the parameters. The difficulty of the puzzle is tuned by increasing or decreasing the number of trailing zeros d. Table 4 summarizes the memory-bound scheme categories and their properties.

5.3 Bandwidth and Network Bound Schemes

Walfish et al. [129] were the first to suggest using bandwidth as a currency to pay for a service to mitigate DoS attacks. They did not propose a puzzle scheme, but introduced the idea of using bandwidth resources to weaken powerful attackers with higher CPU and memory resources than legitimate clients. The proposed system, called *Speak-up*, crowds out attackers by encouraging clients to send higher volumes of traffic for their legitimate requests to be served first. The authors assume that attackers are already dedicating the highest amount of upload bandwidth resources to perform the DoS attack, which prevents them from reacting to the encouragement.

In the following, we describe the different puzzle construction schemes found in the literature that rely either on bandwidth resources or network latency in the evaluation of the puzzle.

5.3.1 Guided Tour Puzzles. Abliz and Tznati [2] introduced the idea of network-bound puzzles to overcome the shortcomings of previous client puzzle schemes, which include parallelizability and computational disparities among clients. Their scheme requires the client to collect tokens from *tour guides* (a pre-specified set of nodes) to be able to solve the puzzle and have their query processed by the service provider. The tour guides are used to introduce a delay between client requests. The authors suggest network latency as a solution to eliminate the disparity in the amount of resources between a powerful adversary and a legitimate client. Non-parallelizability is achieved by the random selection of the next tour guide to be visited at each tour guide stop (hence, the name guided tour) and by requiring the client to submit the hash values from the previous tour guides at each stop. The difficulty is tuned by increasing or decreasing the tour length (number of tour guides) by one, which provides linear puzzle granularity.

5.3.2 Bandwidth Puzzles. Reiter et al. [110] proposed a bandwidth-bound puzzle scheme to validate that a peer did actually expend a certain amount of communication resources and relayed data to peers in a P2P content-distribution system. The main goal is to prevent colluding attackers from earning rewards by claiming that they have exchanged data to each other without actually transferring any data. Their puzzle scheme can be considered relevant to proofs of data possession mechanisms as the solution depends on the possession of the content. Finding the solution is relatively easy for provers who possesses the content, while more difficult for those who do not. Each puzzle is composed of a hash function and a collection of index-sets, where each set contains k random content indices. The verifier constructs a puzzle by hashing the content bits, which are indexed by a pseudorandomly selected index-set and attached together in a certain order. Finding the solution of the puzzle requires determining which of the index-sets is the pre-image of the hash. The puzzles are issued simultaneously to a group of provers and must be solved within a specific time. This prevents a content-holder from colluding with others by solving their puzzles, since he can only solve one puzzle at a time. On a follow up work, Zhang [139] studied the effectiveness of the proposed scheme [110] and provided a lower bound on the number of content bits attackers should possess to be able to defeat the scheme with a definite probability. This lower bound could be used to properly set the puzzle's parameters in real-world systems to improve the overall security.

6 FURTHER DEVELOPMENTS

After discussing the main approaches used in designing the different types of puzzles, it is possible to draw a high-level view of the state-of-the-art in this field. The limitations highlighted in Section 5 could be used to identify future developments, driven by the need to improve the performance and effectiveness of the puzzles. New properties required by the adoption of new technologies could also lead to further advancement in puzzle construction schemes.

6.1 Edge/Fog Architecture

The microservice architecture, enabled by several virtualization techniques such as containers and unikernels, is increasingly used in the cloud environment, as well as in the edge/fog architecture [30]. In this context, bandwidth or network bound puzzle schemes could be very important to mitigate one of the major security concerns, the DDoS attack problem, which aims to destroy the availability of services. Unfortunately, existing schemes described in Section 5.3 are not feasible for this specific scenario due to the high workload required at the server-side to generate the puzzle. For this reason, further research efforts are required to design more efficient schemes with low construction and verification costs suitable for this kind of scenario. Moreover, bandwidth or network bound puzzle schemes are normally used too high in the TCP/IP protocol stack (i.e., application layer), leaving the underlying layers exposed to DDoS attacks. By moving the application of the puzzles to the lower layer, it may be possible to block DDoS attacks in advance, also protecting the upper layers.

6.2 Resource Constraint Devices

The massive adoption of IoT networks has significantly changed the architecture of end-user applications, which is designed to minimize latency in client-server communications and works on devices with limited resources. Many research efforts have been made to ensure the portability of existing puzzle technologies on this new architecture, but some use-cases, such as blockchain-based applications, still require several advancements. The high amount of electricity required to solve some CPU-bound puzzles used in different cryptocurrency schemes has triggered a heavy debate [36], which highlighted the weaknesses of this type of puzzles. Consequently, several new memory-bound schemes have been proposed and used as proof-of-work in some cryptocurrencies [57] to overcome these issues. One of the most successful proposals in this context is CryptoNote [125], a memory-bound scheme designed to ensure the fairness property of the puzzle used as a proof of work in cryptocurrency protocols. However, the CryptoNote scheme has not been sufficiently discussed in the literature. Given the increasing attention it is receiving, important research efforts are needed to verify the soundness of the mathematical properties underlying this scheme, as well as the resistance against the circumvent memory attack. Moreover, some of its properties make the cryptocurrencies based on this scheme appealing to malicious actors, since resource constraint devices can also be used for mining activities [73]. Consequently, the need to modify the characteristics of the puzzles to prevent illegal behavior, as well as to develop valid countermeasures arises, opening new interesting research perspectives.

6.3 Anonymous Networks

Anonymous communications have been introduced to increase the users' privacy within the shared public network environment. Their aim is to provide anonymity between users, apart from content privacy and integrity that are ensured by other technologies [111]. Examples of anonymous networks are Virtual Private Networks (VPNs), Onion routing, web MIXes, peer-to-peer anonymous communications systems, and possibly others. These systems, apart from other attacks that compromise the anonymity, are vulnerable to DDoS attacks that aim to disrupt the service.

Acronym of The Onion Router, the Tor network [38] not only allows users to be anonymous to the website but also the website to be anonymous to the users. This requires the establishment of a circuit that makes use of different levels of encryption. Beyond the privacy problems that users can still be affected to References [69, 119], the Tor network can be subjected to different denial of service attacks. An example of this type of attack is discussed in Reference [14], which proposes CellFlood, a new denial of service attack targeting Tor routers affecting the router's ability to create a new circuit. The authors have shown how a puzzle can be used to allow Tor routers under attack to slow down the attacking hosts, which maintains their ability to manage legitimate client requests. However, the authors used a CPU-bound puzzle, that does not meet the key requirements of the peer-to-peer network puzzles, discussed in Section 4.1.4, most of which are also valid for the Tor Network. In this context, further research efforts are needed to design new puzzle schemes feasible for the anonymous network environment. Bandwidth or network bound puzzle schemes could be considered in this scenario, with the aim of improving the efficiency and fairness of the overall architecture.

6.4 New Generation of Mobile Networks

The advent of the new Generation Wireless System (such as the 5G and, in perspective, 6G) is radically reshaping the business opportunities of mobile network operators, allowing the implementation of several use-cases, such as Vehicular Ad-hoc Networks (VANETs). This novel environment demands the study of new requirements for the development of effective countermeasures to common and new threats. In this context, several authentication schemes have been proposed to protect communications within VANETs. Regardless of their implementation details, all the proposed protocols suffer from DoS attacks that, in the low latency and high bandwidth 5G architecture, can be easily performed. To mitigate this problem, a puzzle-based co-authentication scheme has been presented in Reference [80]. The main goal of this proposal is to make the number of authentication requests that a vehicle can generate in a specific time interval less than or equal to the number of requests that a legitimate vehicle can verify in the same amount of time. The authors modeled their puzzle as a variant of the Hashcash [10] scheme, carefully adjusting the difficulty level to achieve the aforementioned condition. Although promising, this solution has several limitations. First, the proposed countermeasure are effective under the assumption that the attacker has the same computational power of the legitimate vehicles. An attacker equipped with more powerful hardware could solve the puzzles efficiently, easily bypassing the countermeasure. The use of a CPU-bound scheme in 5G networks could also lead to other problems. The energy consumption needed for the computation of the solution could be a serious disadvantage, especially for resource constraint devices typically used in mobile networks. Bandwidth-bound puzzles may also be ineffective in this new network architecture due to the increase in bandwidth available for individual devices in the 5G environment. Further research efforts are needed to identify all these limitations and collect new specifications to design a new generation of puzzle schemes tailored to the peculiarities of the 5G network architecture. Furthermore, while the deployment of 5G is at its beginning, the scientific community already began the design of the next generation of mobile networks: the 6G. This technology, although still in its design stage, is expected to merge the physical and the digital world by providing full support to a large variety of sensors, thus enabling many new use cases [65]. In this context, the security risks will grow as the possible threats cross from the digital to the physical world, magnifying the consequences of possible successful attacks. This scenario increases the need to develop effective defenses and, as already mentioned, puzzles could be very useful to this end.

6.5 Quantum and Post-quantum Era

The potential danger posed to IT security by quantum computing was first established in 1994, by a U.S. mathematician and computer scientist, Peter Shor. He published a quantum computer algorithm [117] able to theoretically break, in a matter of seconds, some of the most used encryption techniques previously assumed secure.

Most of the puzzles are resistant to the threat introduced by this technology. Quantum computing only increases the computational power of an attacker with no effect on puzzles bounded by resources other than the CPU, such as the memory or the network. However, not all CPU-bound puzzles will be compromised by quantum computing. Several works demonstrated that quantum computers are capable of solving complex problems unfeasible for classic computers only by using algorithms that exploit the power of quantum parallelism. For example, a quantum computer cannot be faster than a standard one in multiplications [86]. Quantum computers could be used to efficiently solve some problems underlying the asymmetric cryptography, such as the large prime integer factorization and the discrete logarithm problem, while they could not be as efficient in computing the pre-image of a hash function or in generating a collision. Consequently, the hash-based puzzles can be safely used in the post-quantum era as long as they use a hash function that provides an output with an adequate length, such as sha-2 or sha-3 [86]. Further research efforts are needed in this field to evaluate the security of the non-hash-based puzzles in the quantum world and make them resistant against quantum computation. The first effort in this direction has been made by Brassard et al. in Reference [25], providing a quantum-resistant key establishment scheme based on Merkle puzzle.

6.6 Location-based Services

The new generation of mobile networks, together with the mobile devices market, are also pushing the spreading of a new category of services, called Location-based Services (LBSs). LBSs provide users with accurate and targeted information based on their geographic location, enabling a wide range of use cases, especially in VANETs. In this context, the validation of the real position of a client requesting access to an LBS service becomes of primary importance to ensure the security and stability of the entire system. Several mechanisms, called proof-of-location, have been proposed to solve this problem. The proof-of-location aims to certify the presence of a device in a specific geographical coordinate at a particular instant of time. Puzzles could be used in this context to allow a prover to demonstrate his geographic location to a verifier. Researchers have studied this scenario in References [6, 24], proposing a blockchain-based proof-of-location mechanism to ensure location trustworthiness without compromising user privacy.

7 CONCLUSION

In this article, we have provided an extensive introduction to the concept of *"puzzle"* and presented the evolution of this notion. We have studied the various types of puzzles from different perspectives and classified them based on the way they are applied, their specific context, and the required resources that bound their solving time. Based on our extensive review on the applications of puzzles, we have identified the key requirements that must be satisfied for a puzzle to be applied in a particular field. We have also investigated the influence and impact of puzzles and carved out the elements that hinder their effectiveness in each of the addressed fields. Moreover, we have provided a thorough review of the different types of construction schemes, including CPU-bound, memory-bound, memory-hard, and bandwidth-bound schemes. For each type, we examined the different approaches and techniques used in designing the puzzle and highlighted their distinctive characteristics. We have also identified the limitations and benefits provided by each technique. Further, we have also provided a few research directions that may lead to new insights in the field of puzzles. Finally, this survey, other than being interesting on its own, can also be used as a guideline for designing effective puzzles for a wide range of applications.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their comments and suggestions, which helped improve the quality of the manuscript.

REFERENCES

[1] Martin Abadi, Mike Burrows, Mark Manasse, and Ted Wobber. 2005. Moderately hard, memory-bound functions. *ACM Trans. Internet Technol.* 5, 2 (2005), 299–327.

[2] Mehmud Abliz and Taieb Znati. 2009. A guided tour puzzle for denial of service prevention. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'09)*. IEEE, 279–288.

[3] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. 2017. Scrypt is maximally memory-hard. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 33–62.

[4] Joël Alwen and Björn Tackmann. 2017. Moderately hard functions: Definition, instantiations, and applications. In *Proceedings of the Theory of Cryptography Conference*. Springer, 493–526.

[5] Ghous Amjad, Muhammad Shujaat Mirza, and Christina Pöpper. 2018. Forgetting with puzzles: Using cryptographic puzzles to support digital forgetting. In *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy*. ACM, 342–353.

[6] M. Amoretti, G. Brambilla, F. Medioli, and F. Zanichelli. 2018. Blockchain-based proof of location. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C'18)*. 146–153.

[7] James Aspnes, Collin Jackson, and Arvind Krishnamurthy. 2005. *Exposing Computationally challenged Byzantine Impostors.* Technical Report. Yale University Department of Computer Science.

[8] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. 2014. Proofs of space: When space is of the essence. In *Proceedings of the International Conference on Security and Cryptography for Networks*. Springer, 538–557.

[9] Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. 2000. DOS-resistant authentication with client puzzles. In *Proceedings of the International Workshop on Security Protocols*. Springer, 170–177.

[10] Adam Back. 2002. Hashcash - A denial of service counter-measure. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.8.

[11] Henry S. Baird, Allison L. Coates, and Richard J. Fateman. 2003. Pessimalprint: A reverse turing test. *Int. J. Doc. Anal. Recogn.* 5, 2–3 (2003), 158–163.

[12] Marshall Ball, Alon Rosen, Manuel Sabin, and Prashant Nalini Vasudevan. 2018. Proofs of work from worst-case assumptions. In *Proceedings of the Annual International Cryptology Conference*. Springer, 789–819.

[13] Boaz Barak and Mohammad Mahmoody-Ghidary. 2009. Merkle puzzles are optimal'an O (n2)-query attack on any key exchange from a random oracle. In *Proceedings of the Conference on Advances in Cryptology (CRYPTO'09)*. Springer, 374–390.

[14] Marco Valerio Barbera, Vasileios P Kemerlis, Vasilis Pappas, and Angelos D. Keromytis. 2013. CellFlood: Attacking tor onion routers on the cheap. In *Proceedings of the European Symposium on Research in Computer Security*. Springer, 664–681.

[15] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. 2016. Argon2: New generation of memory-hard functions for password hashing and other applications. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P'16)*. IEEE.

[16] Alex Biryukov and Dmitry Khovratovich. 2016. Egalitarian computing. In *Proceedings of the USENIX Security Symposium*. 315–326.

[17] Alex Biryukov and Dmitry Khovratovich. 2017. Equihash: Asymmetric proof-of-work based on the generalized birthday problem. *Ledger* 2 (2017), 1–30.

[18] Jeremiah Blocki and Hong-Sheng Zhou. 2016. Designing proof of human-work puzzles for cryptocurrency and beyond. In *Proceedings of the Theory of Cryptography Conference*. Springer, 517–546.

[19] M. Blum, L. A. Von Ahn, J. Langford, and N. Hopper. 2000. The captcha project (completely automatic public turing test to tell computers and humans apart). School of Computer Science, Carnegie-Mellon University. Retrieved from http://www.captcha.net.

[20] Carlo Blundo and Stelvio Cimato. 2004. A software infrastructure for authenticated web metering. *Computer* 37, 4 (2004), 28–33.

[21] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable delay functions. In *Proceedings of the Annual International Cryptology Conference*. Springer, 757–788.

[22] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. 2015. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'15)*. IEEE, 104–121.

[23] Nikita Borisov. 2006. Computational puzzles as sybil defenses. In *Proceedings of the 6th IEEE International Conference on Peer-to-Peer Computing*. IEEE Computer Society, 171–176.

[24] M. Amoretti, G. Brambilla, F. Medioli, and F. Zanichelli. 2018. Blockchain-based proof of location. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C'18)*. 146–153.

[25] Gilles Brassard, Peter Høyer, Kassem Kalach, Marc Kaplan, Sophie Laplante, and Louis Salvail. 2011. Merkle puzzles in a quantum world. In *Proceedings of the Conference on Advances in Cryptology (CRYPTO'11)*, Phillip Rogaway (Ed.). Springer, Berlin, 391–410.

[26] Elie Bursztein, Jonathan Aigrain, Angelika Moscicki, and John C. Mitchell. 2014. The end is nigh: Generic solving of text-based CAPTCHAs. In *Proceedings of the 8th USENIX Workshop on Offensive Technologies (WOOT'14)*. USENIX.

[27] Elie Bursztein, Steven Bethard, Celine Fabry, John C. Mitchell, and Dan Jurafsky. 2010. How good are humans at solving CAPTCHAs? A large scale evaluation. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'10)*. IEEE, 399–413.

[28] J.-Y. Cai, Richard J. Lipton, Robert Sedgewick, and A.C.-C. Yao. 1993. Towards uncheatable benchmarks. In *Proceedings of the 8th Annual Structure in Complexity Theory Conference*. IEEE, 2–11.

[29] Jin-Yi Cai, Ajay Nerurkar, and Min-You Wu. 1998. Making benchmarks uncheatable. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS'98)*. IEEE, 216–226.

[30] M. Caprolu, R. Di Pietro, F. Lombardi, and S. Raponi. 2019. Edge computing perspectives: Architectures, technologies, and open security issues. In *Proceedings of the IEEE International Conference on Edge Computing (EDGE'19)*. 116–123. DOI: https://doi.org/10.1109/EDGE.2019.00035

[31] Liqun Chen and Wenbo Mao. 2001. An auditable metering scheme for web advertisement applications. In *Proceedings of the International Conference on Information Security*. Springer, 475–485.

[32] Liqun Chen, Paul Morrissey, Nigel P. Smart, and Bogdan Warinschi. 2009. Security notions and generic constructions for client puzzles. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 505–523.

[33] Bram Cohen and Krzysztof Pietrzak. 2018. Simple proofs of sequential work. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 451–467.

[34] Wei Dai. 1998. B-money proposal. *White Paper.*

[35] Dancho Danchev. 2008. Inside India's CAPTCHA solving economy. Retrieved from https://www.zdnet.com/article/inside-indias-captcha-solving-economy/.

[36] Alex de Vries. 2018. Bitcoin's growing energy problem. *Joule* 2, 5 (2018), 801–805. Retrieved from http://www.sciencedirect.com/science/article/pii/S2542435118301776.

[37] Drew Dean and Adam Stubblefield. 2001. Using client puzzles to protect TLS. In *Proceedings of the USENIX Security Symposium*, Vol. 42.

[38] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. *Tor: The second-generation onion router*. Technical Report. Naval Research Lab, Washington, D.C.

[39] Sujata Doshi, Fabian Monrose, and Aviel D. Rubin. 2006. Efficient memory bound puzzles using pattern databases. In *Proceedings of the International Conference on Applied Cryptography and Network Security*. Springer, 98–113.

[40] John R. Douceur. 2002. The sybil attack. In *Proceedings of the International Workshop on Peer-to-peer Systems*. Springer, 251–260.

[41] Cynthia Dwork, Andrew Goldberg, and Moni Naor. 2003. On memory-bound functions for fighting spam. In *Proceedings of the Annual International Cryptology Conference*. Springer, 426–444.

[42] Cynthia Dwork and Moni Naor. 1992. Pricing via processing or combatting junk mail. In *Proceedings of the Annual International Cryptology Conference*. Springer, 139–147.

[43] Cynthia Dwork, Moni Naor, and Hoeteck Wee. 2005. Pebbling and proofs of work. In *Proceedings of the Annual International Cryptology Conference*. Springer, 37–54.

[44] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. 2015. Proofs of space. In *Proceedings of the Annual Cryptology Conference*. Springer, 585–605.

[45] Ittay Eyal and Emin Gün Sirer. 2018. Majority is not enough: Bitcoin mining is vulnerable. *Commun. ACM* 61, 7 (2018), 95–102.

[46] P. Fairley. 2017. Blockchain world—Feeding the blockchain beast if bitcoin ever does go mainstream, the electricity needed to sustain it will be enormous. *IEEE Spectrum* 54, 10 (Oct. 2017), 36–59.

[47] Wu-chang Feng. 2003. The case for TCP/IP puzzles. In *Proceedings of the ACM SIGCOMM Computer Communication Review*, Vol. 33. ACM, 322–327.

[48] Wu-chi Feng, E. Kaiser, and Antoine Luu. 2005. Design and implementation of network puzzles. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'05)*, Vol. 4. IEEE, 2372–2382.

[49] Matthew K. Franklin and Dahlia Malkhi. 1997. Auditable metering with lightweight security. In *Proceedings of the International Conference on Financial Cryptography*. Springer, 151–160.

[50] Song Gao, Manar Mohamed, Nitesh Saxena, and Chengcui Zhang. 2015. Emerging image game CAPTCHAs for resisting automated and human-solver relay attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 11–20.

[51] Yi Gao, Willy Susilo, Yi Mu, and Jennifer Seberry. 2010. Efficient trapdoor-based client puzzle against DoS attacks. In Network Security, Scott C.-H. Huang, David MacCallum, and Ding-Zhu Du (Eds.). Springer US, 229–249. https://doi.org/10.1007/978-0-387-73821-5_10

[52] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. The bitcoin backbone protocol: Analysis and applications. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 281–310.

[53] Juan A. Garay, Aggelos Kiayias, and Giorgos Panagiotakos. 2017. *Proofs of work for blockchain protocols*. Technical Report. Cryptology ePrint Archive, Report 2017/775.

[54] Virgil D. Gligor. 2003. Guaranteeing access in spite of distributed service-flooding attacks. In *Proceedings of the International Workshop on Security Protocols*. Springer, 80–96.

[55] Gaurav Goswami, Brian M. Powell, Mayank Vatsa, Richa Singh, and Afzel Noore. 2014. FaceDCAPTCHA: Face detection-based color image CAPTCHA. *Future Gen. Comput. Syst.* 31 (2014), 59–68.

[56] Bogdan Groza and Dorina Petrica. 2006. On chained cryptographic puzzles. In *Proceedings of the 3rd Romanian-Hungarian Joint Symposium on Applied Computational Intelligence (SACI'06)*. Citeseer, 25–26.

[57] R. Han, N. Foutris, and C. Kotselidis. 2019. Demystifying crypto-mining: Analysis and optimizations of memory-hard pow algorithms. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'19)*. 22–33. DOI: https://doi.org/10.1109/ISPASS.2019.00011

[58] José María Gómez Hidalgo and Gonzalo Alvarez. 2011. Captchas: An artificial intelligence application to web security. In *Advances in Computers*. Vol. 83. Elsevier, 109–181.

[59] Dennis Hofheinz, Tibor Jager, Dakshita Khurana, Amit Sahai, Brent Waters, and Mark Zhandry. 2016. How to generate and use universal samplers. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 715–744.

[60] Kuo-Feng Hwang, Cian-Cih Huang, and Geeng-Neng You. 2012. A spelling-based CAPTCHA system by using click. In *Proceedings of the International Symposium on Biometrics and Security Technologies*. IEEE.

[61] Markus Jakobsson and Ari Juels. 1999. Proofs of work and bread pudding protocols. In *Secure Information Networks*. Springer, 258–272.

[62] Yves Igor Jerschow and Martin Mauve. 2010. Offline submission with rsa time-lock puzzles. In *Proceedings of the 10th IEEE International Conference on Computer and Information Technology (CIT'10)*. IEEE, 1058–1064.

[63] Ari Juels and John G. Brainard. 1999. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'99)*, Vol. 99. 151–165.

[64] Ed Kaiser and Wu-chang Feng. 2008. mod kapow: Protecting the web with transparent proof-of-work. In *Proceedings of the INFOCOM Workshops*. IEEE, 1–6.

[65] Raimo Kantola. 2019. 6G network needs to support embedded trust. In *Proceedings of the 14th International Conference on Availability, Reliability, and Security (ARES'19)*. ACM, New York, NY. DOI: https://doi.org/10.1145/3339252.3341498

[66] Ghassan O. Karame and Srdjan Čapkun. 2010. Low-cost client puzzles based on modular exponentiation. In *Proceedings of the European Symposium on Research in Computer Security*. Springer, 679–697.

[67] Jonathan Katz, Andrew Miller, and Elaine Shi. 2014. Pseudonymous broadcast and secure computation from cryptographic puzzles. Cryptology ePrint Archive, Report 2014/857. https://eprint.iacr.org/2014/857

[68] Rohit Ashok Khot and Kannan Srinathan. 2009. iCAPTCHA: Image tagging for free. In *Proceedings of the Conference on Usable Software and Interface Design*, Vol. 2. 26.

[69] Massimo La Morgia, Alessandro Mei, Simone Raponi, and Julinda Stefa. 2018. Time-zone geolocation of crowds in the dark web. In *Proceedings of the IEEE 38th International Conference on Distributed Computing Systems (ICDCS'18)*. IEEE, 445–455.

[70] Leslie Lamport. 1981. Password authentication with insecure communication. *Commun. ACM* 24, 11 (1981), 770–772.

[71] Ben Laurie and Richard Clayton. 2004. Proof-of-work proves not to work; version 0.2. In *Proceedings of the Workshop on Economics and Information Security*.

[72] Jonathan Lazar, Jinjuan Feng, Tim Brooks, Genna Melamed, Brian Wentz, Jon Holman, Abiodun Olalere, and Nnanna Ekedebe. 2012. The soundsright CAPTCHA: An improved approach to audio human interaction proofs for blind users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2267–2276.

[73] E. Le Jamtel. 2018. Swimming in the monero pools. In *Proceedings of the 11th International Conference on IT Security Incident Management IT Forensics (IMF'18)*. 110–114. DOI: https://doi.org/10.1109/IMF.2018.00016

[74] Charles Lee. 2011. Litecoin-open source p2p digital currency. Retrieved from https://litecoin.org/.

[75] Arjen Klaas Lenstra, Hendrik Willem Lenstra, and László Lovász. 1982. Factoring polynomials with rational coefficients. *Math. Ann.* 261, 4 (1982), 515–534.

[76] Frank Li, Prateek Mittal, Matthew Caesar, and Nikita Borisov. 2012. SybilControl: Practical sybil defense with computational puzzles. In *Proceedings of the 7th ACM Workshop on Scalable Trusted Computing*. ACM, 67–78.

[77] Mark D. Lillibridge, Martin Abadi, Krishna Bharat, and Andrei Z. Broder. 2001. Method for selectively restricting access to computer systems. U.S. Patent 6,195,698.

[78] Huijia Lin, Rafael Pass, and Pratik Soni. 2017. Two-round and non-interactive concurrent non-malleable commitments from time-lock puzzles. In *Proceedings of the IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS'17)*. IEEE, 576–587.

[79] Debin Liu and L. Jean Camp. 2006. Proof of work can work. In *5th Annual Workshop on the Economics of Information Security (WEIS'16), Robinson College, University of Cambridge, England, UK, June 26-28, 2006*. http://weis2006.econinfosec.org/docs/50.pdf.

[80] P. Liu, B. Liu, Y. Sun, B. Zhao, and I. You. 2018. Mitigating dos attacks against pseudonymous authentication through puzzle-based co-authentication in 5G-VANET. *IEEE Access* 6 (2018), 20795–20806.

[81] Quanquan Liu. 2017. Red-blue and standard pebble games: complexity and applications in the sequential and parallel models. Ph.D. Dissertation. Massachusetts Institute of Technology.

[82] Sam Loyd. 1959. *Mathematical puzzles*. Vol. 1. Courier Corporation.

[83] Loi Luu, Ratul Saha, Inian Parameshwaran, Prateek Saxena, and Aquinas Hobor. 2015. On power splitting games in distributed computation: The case of bitcoin pooled mining. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium (CSF'15)*. IEEE, 397–411.

[84] Miao Ma. 2005. Mitigating denial of service attacks with password puzzles. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, Vol. 2. IEEE, 621–626.

[85] Mohammad Mahmoody, Tal Moran, and Salil Vadhan. 2013. Publicly verifiable proofs of sequential work. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*. ACM, 373–388.

[86] Vasileios Mavroeidis, Kamer Vishi, Mateusz D. Zych, and Audun Jøsang. 2018. The impact of quantum computing on present cryptography. *Int. J. Adv. Comput. Sci. Appl.* 9, 3 (2018).

[87] Timothy J. McNevin, Jung-Min Park, and Randolph Marchany. 2004. pTCP: A client puzzle protocol for defending against resource exhaustion denial of service attacks. Virginia Tech Univ., Dept. Elect. Comput. Eng., Blacksburg, VA, Technical Report TR-ECE-04-10.

[88] Ralph C. Merkle. 1978. Secure communications over insecure channels. *Commun. ACM* 21, 4 (1978), 294–299.

[89] Hendrik Meutzner, Santosh Gupta, and Dorothea Kolossa. 2015. Constructing secure audio captchas by exploiting differences between humans and machines. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 2335–2338.

[90] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. 2014. Permacoin: Repurposing bitcoin work for data preservation. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'14)*. IEEE, 475–490.
[91] Andrew Miller, Ahmed Kosba, Jonathan Katz, and Elaine Shi. 2015. Nonoutsourceable scratch-off puzzles to discourage bitcoin mining coalitions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 680–691.
[92] Andrew Miller and Joseph J. LaViola Jr. 2014. Anonymous byzantine consensus from moderately hard puzzles: A model for bitcoin. Retrieved from http://nakamotoinstitute.org/research/anonymous-byzantine-consensus.
[93] Greg Mori and Jitendra Malik. 2003. Recognizing objects in adversarial clutter: Breaking a visual CAPTCHA. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Vol. 1. IEEE.
[94] Robert Moskowitz, Pekka Nikander, Petri Jokela, and Thomas Henderson. 2008. *Host identity protocol*. Technical Report.
[95] Marti Motoyama, Kirill Levchenko, Chris Kanich, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage. 2010. Re: CAPTCHAs-Understanding CAPTCHA-solving services in an economic context. In *Proceedings of the USENIX Security Symposium*, Vol. 10. 3.
[96] Satoshi Nakamoto. 2019. Bitcoin: A peer-to-peer electronic cash system. Manubot. Retrieved from https://git.dhimmel.com/bitcoin-whitepaper/.
[97] Moni Naor. 1996. Verification of a human in the loop or identification via the turing test. Retrieved from http://www.wisdom.weizmann.ac.il/~naor/PAPERS/humanabs.html.
[98] Arvind Narayanan and Jeremy Clark. 2017. Bitcoin's academic pedigree. *Commun. ACM* 60, 12 (2017), 36–45.
[99] M. A. Noureddine, A. M. Fawaz, A. Hsu, C. Guldner, S. Vijay, T. Başar, and W. H. Sanders. 2019. Revisiting client puzzles for state exhaustion attacks resilience. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'19)*. 617–629.
[100] Erik Nygren, Samuel Erb, Alex Biryukov, Dmitry Khovratovich, and Ari Juels. 2016. TLS client puzzles extension. *Internet Engineering Task Force, Technical Report* (Dec. 2016).
[101] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algor.* 51, 2 (2004), 122–144.
[102] Bryan Parno, Dan Wendlandt, Elaine Shi, Adrian Perrig, Bruce Maggs, and Yih-Chun Hu. 2007. Portcullis: Protecting connection setup from denial-of-capability attacks. *ACM SIGCOMM Comput. Commun. Rev.* 37, 4 (2007), 289–300.
[103] Rafael Pass, Lior Seeman, and Abhi Shelat. 2017. Analysis of the blockchain protocol in asynchronous networks. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 643–673.
[104] Colin Percival. 2009. Stronger key derivation via sequential memory-hard functions. BSDCan.
[105] Heiner Perrey, Osman Ugus, and Dirk Westhoff. 2011. WiSec'2011 poster: Security enhancement for bluetooth low energy with Merkle's puzzle. *ACM SIGMOBILE Mobile Comput. Commun. Rev.* 15, 3 (2011), 45–46.
[106] Krzysztof Pietrzak. 2018. Simple verifiable delay functions.*IACR Cryptol. ePrint Arch.* 2018 (2018), 627.
[107] Rajesh Ramanathan, Amritansh Raghav, and Craig M. Combel. 2013. Spam reduction in real time communications by human interaction proof. U.S. Patent 8,495,727.
[108] Jothi Rangasamy, Douglas Stebila, Lakshmi Kuppusamy, Colin Boyd, and Juan Gonzalez Nieto. 2011. Efficient modular exponentiation-based puzzles for denial-of-service protection. In *Proceedings of the International Conference on Information Security and Cryptology*. Springer, 319–331.
[109] Amar Rasheed and Rabi Mahapatra. 2011. Key predistribution schemes for establishing pairwise keys with a mobile sink in sensor networks. *IEEE Trans. Parallel Distrib. Syst.* 22, 1 (2011), 176–184.
[110] Michael K. Reiter, Vyas Sekar, Chad Spensky, and Zhenghao Zhang. 2009. Making peer-assisted content distribution robust to collusion using bandwidth puzzles. In *Proceedings of the International Conference on Information Systems Security*. Springer, 132–147.
[111] Jian Ren and Jie Wu. 2010. Survey on anonymous communications in computer networks. *Comput. Commun.* 33, 4 (2010), 420–431. DOI: https://doi.org/10.1016/j.comcom.2009.11.009
[112] Ling Ren and Srinivas Devadas. 2016. Proof of space from stacked expanders. In *Proceedings of the Theory of Cryptography Conference*. Springer, 262–285.
[113] Ling Ren and Srinivas Devadas. 2017. Bandwidth hard functions for ASIC resistance. In *Proceedings of the Theory of Cryptography Conference*. Springer, 466–492.
[114] Ronald L. Rivest and Adi Shamir. 1996. PayWord and micromint: Two simple micropayment schemes. In *Proceedings of the International Workshop on Security Protocols*. Springer, 69–87.
[115] Ronald L. Rivest, Adi Shamir, and David A. Wagner. 1996. Time-lock puzzles and timed-release crypto. Massachusetts Institute of Technology.
[116] Hosam Rowaihy, William Enck, Patrick McDaniel, and Tom La Porta. 2007. Limiting sybil attacks in structured p2p networks. In *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM'07)*. IEEE, 2596–2600.

[117] Peter W. Shor. 1994. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the Annual Symposium on Foundations of Computer Science*. IEEE, 124–134.
[118] Douglas Stebila and Berkant Ustaoglu. 2009. Towards denial-of-service-resilient key agreement protocols. In *Proceedings of the Australasian Conference on Information Security and Privacy*. Springer, 389–406.
[119] Yixin Sun, Anne Edmundson, Laurent Vanbever, Oscar Li, Jennifer Rexford, Mung Chiang, and Prateek Mittal. 2015. RAPTOR: Routing attacks on privacy in tor. In *Proceedings of the 24th USENIX Security Symposium (USENIX'15)*. 271–286.
[120] Nick Szabo. 2008. Bit gold, unenumerated. blogspot.com (Mar. 29, 2006) Internet Archive. Retrived from http://unenumerated.blogspot.com/2005/12/bit-gold.html.
[121] Qiang Tang and Arjan Jeckmans. 2010. On non-parallelizable deterministic client puzzle scheme with batch verification modes. Centre for Telematics and Information Technology, University of Twente.
[122] Suratose Tritilanunt, Colin Boyd, Ernest Foo, and Juan Manuel González Nieto. 2007. Toward non-parallelizable client puzzles. In *Proceedings of the International Conference on Cryptology and Network Security*. Springer, 247–264.
[123] John Tromp. 2015. Cuckoo cycle: A memory bound graph-theoretic proof-of-work. In *Proceedings of the International Conference on Financial Cryptography and Data Security*. Springer, 49–62.
[124] Florian Tschorsch and Björn Scheuermann. 2016. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IEEE Commun. Surveys Tutor.* 18, 3 (2016), 2084–2123.
[125] Nicolas Van Saberhagen. 2013. CryptoNote v 2.0.
[126] Luis Von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. 2003. CAPTCHA: Using hard AI problems for security. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 294–311.
[127] Luis Von Ahn, Benjamin Maurer, Colin McMillen, David Abraham, and Manuel Blum. 2008. recaptcha: Human-based character recognition via web security measures. *Science* 321, 5895 (2008), 1465–1468.
[128] David Wagner. 2002. A generalized birthday problem. In *Proceedings of the Annual International Cryptology Conference*. Springer, 288–304.
[129] Michael Walfish, Mythili Vutukuru, Hari Balakrishnan, David Karger, and Scott Shenker. 2010. DDoS defense by offense. *ACM Trans. Comput. Syst.* 28, 1 (2010), 3.
[130] Wenbo Wang, Dinh Thai Hoang, Zehui Xiong, Dusit Niyato, Ping Wang, Peizhao Hu, and Yonggang Wen. 2018. A survey on consensus mechanisms and mining management in blockchain networks. Retrieved from http://arxiv.org/abs/1805.02707.
[131] XiaoFeng Wang and Michael K. Reiter. 2003. Defending against denial-of-service attacks with puzzle auctions. In *Proceedings of the Symposium on Security and Privacy*. IEEE, 78–92.
[132] XiaoFeng Wang and Michael K. Reiter. 2004. Mitigating bandwidth-exhaustion attacks using congestion puzzles. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS'04)*. ACM, New York, NY, 257–267. DOI: https://doi.org/10.1145/1030083.1030118
[133] Brent Waters, Ari Juels, J. Alex Halderman, and Edward W. Felten. 2004. New client puzzle outsourcing techniques for DoS resistance. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*. ACM, 246–256.
[134] Dirk Westhoff and Frederik Armknecht. 2011. Method for electing aggregator nodes in a network. U.S. Patent 7,907,548.
[135] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.
[136] Yi Xu, Gerardo Reynaga, Sonia Chiasson, Jan-Michael Frahm, Fabian Monrose, and Paul C. van Oorschot. 2012. Security and usability challenges of moving-object CAPTCHAs: Decoding codewords in motion. In *Proceedings of the USENIX Security Symposium*. 49–64.
[137] Jeff Yan and Ahmad Salah El Ahmad. 2008. A low-cost attack on a microsoft CAPTCHA. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*. ACM, 543–554.
[138] Jeff Yan and Ahmad Salah El Ahmad. 2008. Usability of CAPTCHAs or usability issues in CAPTCHA design. In *Proceedings of the 4th Symposium on Usable Privacy and Security*. ACM, 44–52.
[139] Zhenghao Zhang. 2012. A new bound on the performance of the bandwidth puzzle. *IEEE Trans. Info. Forensics Secur.* 7, 2 (2012), 731–742.

Received September 2019; revised February 2020; accepted April 2020