IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received May 3, 2021, accepted May 13, 2021, date of publication May 19, 2021, date of current version May 18, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3082037

Client Puzzle Protocols as Countermeasure against Automated Threats to Web Applications

VLADIMIR BOSTANOV[1]

[1]SySS GmbH, Schaffhausenstr. 77, D-72072 Tübingen, Germany (e-mail: vladimir.bostanov@syss.de, vladimir.bostanov@bug.8yz.de)

Corresponding author: Vladimir Bostanov (e-mail: vladimir.bostanov@syss.de, vladimir.bostanov@bug.8yz.de).

This study was supported by the Research & Development Fund of SySS GmbH.

ABSTRACT Proof-of-work (PoW) schemes implemented in client puzzle protocols (CPPs) have been proposed as a protection from Denial-of-Service (DoS) attacks against internet facing servers. A CPP designed to thwart attacks against a certain client-server protocol is layered independently on top of this protocol or is integrated into it. Such a general solution requires a great deal of standardization. On the other hand, different web applications that may also become targets of DoS attacks can be protected by different schemes, which greatly reduces standardization requirements and makes implementations substantially easier. In the the present study, we discuss the utility of CPPs as a practical layer of protection of web applications against DoS and other automated threat events. We define several requirements that must be met by such CPPs, and we propose a general concept and a particular PoW algorithm that fulfills these requirements. The general concept includes recursive definition of sub-puzzles and partial server-side solution verification. The proposed PoW algorithm is based on this concept and on hash inversion/collision tasks. We also introduce a few prototype implementations of this algorithm in JavaScript, WebAssembly, Python, and C, and we present the results of some benchmark tests comparing the performances of these implementations on different hardware. These results show that CPPs can provide an effective layer of mitigation against certain automated threats to web applications.

INDEX TERMS proof of work, client puzzle protocol, automated threat, web application, denial of service

I. INTRODUCTION

PROOF OF WORK (PoW) as a concept has become widely known as a mechanism for achieving consensus in cryptocurrency systems. For instance, a block of transactions added by a miner node to the Bitcoin blockchain needs to include a hash of the block content [1], [2]. In order to be accepted by all other nodes, the hash must fulfill a certain condition (“meet a target”), which can only be achieved by trial and error, i.e. by investing a certain amount of computational resources and power.

Finding a hash that meets a preset target is an example of a puzzle [3]. Long before rising to prominence with the Bitcoin boom, PoW schemes implemented in client puzzle protocols (CPPs) have been proposed as a deterrent to SPAM [4], [5] and as a protection from Denial-of-Service (DoS) attacks against internet facing servers [6], [7]. In the latter case, the client must solve a puzzle in order to be granted access to a service or data provided by the server. An attacker is thus forced to invest a certain amount of computational resources and time for each request, which limits the request rate and prevents the depletion of server resources.

Various PoW schemes have been proposed. Some of the most popular underlying puzzles are based on prime modulo operations [4], [8]–[10], partial hash inversions [5]–[7], [11], [12], and memory intensive computations [13]–[17]. (See [3] for a recent review of puzzle types; for a review of PoW schemes applied to blockchain technology, see [18].)

A CPP designed to thwart DoS attacks against a certain client-server protocol (e.g. TCP, TLS, SMTP, etc.) is layered independently on top of this protocol or is integrated into it [6], [19]–[21]. Such a general solution requires a great deal of standardization.

A web application can also be a target of a DoS attack which is not based on depletion of system-wide resources, but leads to exhaustion of application-specific resources such as file system, memory, processes, threads, CPU, and human or financial resources. The server usually remains unaffected, while the application may be affected as a whole, or the attack may be against individual users such as account lockout.

DoS attacks against web applications have been described as a subset of the broader category of "threat events to web applications undertaken using automated actions". The latter definition was taken from "OWASP Automated Threat Handbook Web Applications" [22]. This handbook provides a list of 21 automated threat events, including, e.g., brute force credential cracking, automated account creation, CAPTCHA defeat, credit card cracking, DoS, etc.. Note that, in the case of web applications, DoS may be caused by depletion of human or other resources not related to computer software or hardware rather than by exhaustion of, e.g., server memory, CPU power, etc.. For example, a tour operator may accept online customer requests via a web form. If these requests are processed by human employees, an attacker could flood them with fake requests by launching an automated form submission attack. Thus, the tour operator employees would be unable to find and process the real customer requests lost in the large quantity of fake ones. With a little more effort (by, e.g., using real people's addresses, etc. in submissions), the attacker could make it very hard even for site and database administrators to separate real from fake requests.

To the best of our knowledge, there has been virtually no academic research on CPPs designed to protect web applications from automated threat [3]. A possible exception is represented by a recent US patent proposing a CPP based on a partial hash inversion puzzle as mitigation against DoS attacks [23]. However, such a simple scheme cannot provide an adequate protection, because the time and energy invested in solving the puzzle can be reduced dramatically by parallel processing performed with special hardware such as graphics processing unit (GPU) and field-programmable gate array (FPGA) clusters [2]. The problem is that a CPP designed to protect web applications must assume that the puzzle is solved by a browser script running on a desktop or mobile device equipped with an ordinary, consumer electronics CPU. An attacker, however, can use a compiled program (written in, e.g., the C language) operating commercially available, not too expensive GPUs or FPGAs to solve the puzzle in a tiny fraction of the time needed by a browser script using only the available CPU resources of the average user's device.

In the the present study, we discuss the utility of CPPs as a practical layer of protection of web applications against some automated threat events. We address the parallelization problem formulated above by defining several requirements that must be met by such CPPs, and we propose a general concept and a particular PoW algorithm that fulfills these requirements. The general concept includes recursive definition of sub-puzzles and partial server-side solution verification. The proposed PoW algorithm is based on this concept and on hash inversion/collision tasks. We also introduce a few prototype implementations of this algorithm in JavaScript, WebAssembly, Python, and C, and we present the results of some benchmark tests comparing the performances of these implementations on different hardware.

It is important to point out, that there is *no* need of a single standard CPP supported by all web applications. Unlike servers and internet protocols, different web applications may be protected by different CPPs based on different PoW algorithms. This allows for great diversity and flexibility and makes the practical implementation much easier than it is for the underlying protocols, TCP, TLS, and HTTP.

II. GENERAL CONCEPT

A CPP designed to protect web applications against automated threats should be based on an algorithm that can be easily implemented in a browser-supported language – JavaScript or WebAssembly. More specifically, an implementation in another language like, e.g. C, or Assembly, must *not* bring a crucial advantage in terms of computational speed, because such an advantage can be used by an attacker to defeat the protection offered by the CPP. For the same reason, the algorithm should also be nonparalellizable, or at least, parallelization should not reduce substantially the average time needed to solve the puzzle [3]. In order to protect the server from DoS attacks by flooding with wrong solutions, the CPP asymmetry requirement must be fulfilled: the time needed for server-side verification should be a tiny fraction of the time needed by the client to provide a solution [3]. For the same reason, the amount of data exchanged between the client and the server should not be too large.

In order to address all these requirements, let us consider a general concept based on the following two principles. First, the client must solve a sequence of $\mathcal{N}$ easy sub-puzzles $\mathcal{P}_n$, $n = 1 \ldots \mathcal{N}$, instead of a single difficult puzzle. However, only the initial sub-puzzle $\mathcal{P}_0$ in the sequence is posed by the server. Each of the following problems $\mathcal{P}_n$ is generated from the initial problem $\mathcal{P}_0$ and the solution S_{n-1} to the previous one. To be more specific,

$$\mathcal{P}_0: \quad \wp(\mathcal{K}, 0) \rightsquigarrow S_0, \tag{1a}$$

$$\mathcal{P}_n: \quad \wp(\mathcal{K}, S_{n-1}) \rightsquigarrow S_n, \quad n = 1 \ldots \mathcal{N} - 1, \tag{1b}$$

where $\wp$ denotes the specific type of problem and $\mathcal{K}$ is a set of parameters provided by the server.

We will further use the word "puzzle" to refer to the whole sequence of sub-puzzles. After solving the puzzle, the client sends the solution to the server. The server, however, must *not* verify the whole solution. Only one sub-solution needs to be verified, in order to impose the desired computational cost on a potential attacker. An attacker who cheats regularly on k out of the $\mathcal{N}$ sub-puzzles will do $k/\mathcal{N}$ less work, but will also fail on average on $k/\mathcal{N}$ of the puzzles and will have to solve the same number of new ones in order to perform the same attack as someone who does not cheat at all. Thus, by introducing the sub-puzzle principle, we achieve a client/server asymmetry quotient of $\mathcal{N}$.

CPPs typically use mathematical problems that can only be solved by trial and error. The ith trial may involve the computation of $y^i = f(\mathcal{K}, x^i)$, where f is some function, and x^i is a set of arguments defining the ith trial. A well known simple example is $f(\mathcal{K}, x^i) = h(\mathcal{K} \| i)$, where h is a hash function,

$\mathcal{K}$ is a string provided by the server or composed by the client according to certain rules, and $\|$ denotes concatenation. The "hashcash" algorithm [5], [7] used in the bitcoin PoW scheme [1] defines the puzzle solution m by $h(\mathcal{K}\|m) < \mathcal{H}$, where $\mathcal{H}$ is the "target", a predefined constant.

The second principle of our general concept is somewhat similar to the first one and it concerns the structure of the sub-puzzle. Let us consider the following recursive definition,

$$y^0 = y^1 = \ldots = y^{\ell-1} = 0, \tag{2a}$$
$$y^i = f(\mathcal{K},\, y^{i-\ell}, y^{i-\ell+1}, \ldots y^{i-1}), \quad i \geq \ell. \tag{2b}$$

Our sub-puzzle will be defined by some (yet unspecified) additional requirements imposed on y^i. A full verification of a solution y^m would include the replication by the server of all $m \geq 2\ell$ iterations completed by the client. However, again, this is not necessary when the goal is to thwart automated attacks against a web application. Instead of performing full verification, the server may verify only y^m and y^q, where q is randomly chosen among $m{-}\ell,\ m{-}\ell{+}1,\ \ldots m{-}1$. Such a partial verification allows the attacker to start the computations from a set of initial values that are different from those defined by (2a). But as long as the complexity of the problem does not depend on the choice of a starting point, this does not reduce the computational cost. It allows, however, for some parallelization. If a randomly chosen y^q were not verified, the attacker could completely ignore the recursion and perform an arbitrary number of parallel computations of y^ℓ with randomly chosen $y^0, y^1, \ldots y^{\ell-1}$. But the verification of y^q renders futile any cheating strategy that would avoid the completion of at least ℓ iterations. This leaves the attacker with the option to start several parallel iteration sequences. If the average number of iterations μ needed to find a solution is large, such parallelization will effectively speed up the computations. If, however, μ is small enough, parallelization will not bring any advantage (or will even increase the computational cost). Thus, if μ is chosen to be not too large, the partial verification of each sub-solution will be sufficient, which yields an asymmetry quotient of $\mu/2$ per sub-puzzle.

Note, however, that choosing a small μ reduces both the computational cost of the puzzle and the asymmetry quotient. Hence, a large enough number of sub-puzzles $\mathcal{N}$ should be chosen in order to compensate for this effect. The overall client/server asymmetry quotient $\mathcal{A}$ is given by

$$\mathcal{A} = \frac{\mu\mathcal{N}}{2}. \tag{3}$$

Sub-puzzles were first introduced by [6] and [11], however, without the notion of partial verification.

III. RECURSIVE HASH PUZZLES

Let us consider more closely two types of puzzles: partial hash inversion and hash collision. Specifying a hash function and applying the general principles stated above, we formulate the two problems as follows. Let $H_\mathcal{B}(K, M)$ denote the first $\mathcal{B}$ bits of the hash-based message authentication code (HMAC [24], [25]) of the message M signed with the key K, using, e.g., the SHA-256 hash function [26], [27]. We define the hash sequence h_n^i for the nth sub-puzzle recursively:

$$h_n^i = 0, \quad \text{for } 0 \leq i < \ell, \tag{4a}$$
$$h_n^i = H_\mathcal{B}(K_n,\, M_n^i), \quad \text{for } i \geq \ell, \tag{4b}$$
$$M_n^i = h_n^{i-\ell}\|h_n^{i-\ell+1}\| \ldots h_n^{i-1}\,\|\,\mathcal{R}, \tag{4c}$$
$$K_0 = \mathcal{K}\|0\|0, \tag{4d}$$
$$K_n = \mathcal{K}\|n\|S_{n-1}, \quad \text{for } 0 < n < \mathcal{N}, \tag{4e}$$

where $\mathcal{K}$ and $\mathcal{R}$ are constants provided by the server and S_n is the solution to the nth sub-puzzle. Then the partial hash inversion problem is defined by

$$S_n = h_n^m < \mathcal{H}, \quad m \geq 2\ell, \quad \mathcal{H} = \text{const}, \tag{5}$$

i.e., a message M composed by a constant string and the hashes resulting from the last ℓ iterations should be found such that its hash is smaller than a predefined constant $\mathcal{H}$ (the target). The hash collision problem is defined by

$$S_n = h_n^m = h_n^k, \quad m > k \geq 2\ell, \tag{6}$$

i.e., two different messages should be found whose hashes are equal.

Note that both tasks require that at least ℓ iterations be performed before even staring to look for a solution. Note also that the size of the constant string $\mathcal{R}$ directly affects the computational cost of each iteration and, hence, of the whole puzzle.

Appendix A provides a formal pseudocode description and a schematic flowchart representation of the proposed algorithm.

IV. PROBABILITY DISTRIBUTIONS

Now, let us examine the probability distributions governing the processes of finding solutions to each of the two problems. In the case of partial hash inversion, at each iteration, the probability p of hitting the target is

$$p = (N - \mathcal{H})/N, \tag{7}$$

where $N = 2^\mathcal{B}$ is the size of the hash space (e.g. $N = 2^{24}$ if the first 24 bits of the HMAC function are used). The probability of *not* hitting the target in m trials is $(1{-}p)^m$, and hence the probability of finding a solution after at least m iterations is $P_m = 1 - (1-p)^m$. Using a continuous approximation to this distribution with cumulative distribution function (CDF) $P(x)$ helps us find the mean number of iterations μ needed for finding a solution and the corresponding standard deviation σ:

$$P(x) = 1 - (1-p)^x = 1 - e^{\ln(1-p)x}, \tag{8a}$$

$$\mu = \int_0^\infty x\,dP = \frac{-1}{\ln(1-p)} \approx \frac{1}{p}, \quad \text{for } p \ll 1, \tag{8b}$$

$$\sigma = \left(\int_0^\infty x^2\,dP - \mu^2\right)^{\frac{1}{2}} = \mu \approx \frac{1}{p}, \quad \text{for } p \ll 1. \tag{8c}$$

In our second case, the probability of completing m trials *without* a collision is [28], [29]:

$$\frac{N!}{N^m(N-m)!)} \approx \exp\left(-\frac{m^2}{2N}\right), \quad \text{for } m \ll N. \tag{9}$$

Using, again, a continuous CDF, we obtain:

$$P(x) \approx 1 - \exp\left(-\frac{x^2}{2N}\right), \tag{10a}$$

$$\mu = \int_0^\infty x\,dP = \sqrt{\frac{\pi N}{2}}, \tag{10b}$$

$$\sigma = \left(\int_0^\infty x^2\,dP - \mu^2\right)^{\frac{1}{2}} = \sqrt{\frac{(4-\pi)N}{2}} \approx \frac{\mu}{2}. \tag{10c}$$

Thus, given the same mean, the standard deviation in the case of partial hash inversion is almost twice as large as the corresponding standard deviation in the hash collision task.

Note that the initial number of iterations ℓ completed before starting to look for a solution should be added to the corresponding expression for μ in order to obtain the actual mean number of trials necessary for finding a solution to a sub-puzzle.

V. SOFTWARE IMPLEMENTATION

In order to test the proposed PoW algorithm, we developed a prototype JavaScript/PHP implementation of a simple CPP based on it. A server-side PHP script generates a new puzzle by setting the type of puzzle $\wp$ (inversion or collision), the number of sub-puzzles $\mathcal{N}$, the number of HMAC bits $\mathcal{B}$, the initial key $\mathcal{K}$ (4d), the recursion depth ℓ (2b, 4c), the size of the constant string $\mathcal{R}$ (4c) determining the computational cost of each iteration, and (in the case of partial hash inversion) the target $\mathcal{H}$ (5). The puzzle is solved by client-side JavaScript and the PoW is sent back to the server. Another PHP script performs partial verification as described above and, in case of success, grants the client an authorization token providing access to the requested resource.

The PoW size is about 2ℓ 32-bit words, or 8ℓ bytes, for the partial hash inversion case and about twice as much for collisions. For $\ell = 1000$ (which is a realistic setting), this yields about 8 kB and 16 kB, respectively. This size is not too large for a request sent by a client script in a contemporary web application. Actually, it is quite reasonable.

The small proof size is achieved by sending the PoW in two steps, as described in Appendix A. First, only the solutions S_n to the sub-puzzles are sent to the server. The size of this part is small: about $4\mathcal{N}$ bytes. Second, the server chooses a random sub-puzzle and requires the necessary PoW sequence(s) for the verification of its solution. The size of this part is 8ℓ bytes for partial hash inversions and 16ℓ bytes for collisions. (Actually, in the current prototype implementation, the size is approximately 33% larger, because the PoW is base64-encoded, which, of course, would not be the case in a real-world implementation.)

In the implementation of the collision problem, a memory-intensive search is avoided, by using a boolean array whose indices are the possible hash values (array I in Appendix A). At each iteration, the program computes a hash and checks whether the array element with the corresponding index has a value of `True` or `False`, the latter being the initial value. If `True`, a solution has been found; else, the value of the array element is changed to `True` and the program proceeds with the next iteration. This procedure is formally described in Appendix A.

Additionally, benchmark routines comprising loops with preset, fixed number of iterations performing only the computationally intensive PoW operations were written in JavaScript, Python, C, and WebAssembly. The WebAssembly code was obtained from the C source using the Emscripten compiler [30]. Benchmark tests measuring mean hash rates were performed on different platforms.

All source code is available under the GNU General Public License (GPL) on GitHub [31].

VI. BENCHMARK TEST RESULTS

Some of the test results are presented in Table 1. These were obtained with number of HMAC bits $\mathcal{B} = 24$, recursion depth $\ell = 1000$, and HMAC message size (controlled by $\mathcal{R}$ in 4c) about 40 kB (10 000 32-bit words). These values of $\mathcal{B}$ and ℓ prevent taking advantage through parallelization and provide a reasonable asymmetry quotient (3) as discussed in Section II. (In the case of partial hash inversion, an appropriate target $\mathcal{H}$ should also be set according to the equations in Section IV, specifically (7) and (8b).) The chosen message size puts enough computational load on each iteration to reduce the number of loop repetitions, thus minimizing the advantage of a compiled (e.g. C) versus an interpreted (JavaScript) implementation.

Each mean hash rate presented in Table 1 was obtained from a sample of 100 trials, each trial comprising 1 000 iterations. In order to simulate recursive algorithm execution, *no* parallelization was used, i.e. all iterations were computed strictly sequentially, and also each new trial was started only after completion of the previous one. Each hash rate was obtained by simply dividing 1 000 by the execution time of the corresponding trial.

CPU/Chipset (OS)	Implementation	kHash/s	SEM
i7-8550U (Ubuntu 18.4)	JS, Firefox	3.0	0.03
	JS, Chrome	7.0	0.05
	WASM, Firefox	4.2	0.03
	WASM, Chrome	3.7	0.02
	C	7.6	0.03
	Python	10.6	0.07
Qualcomm Snapdragon 630 (Android 9)	JS, Chrome	3.7	0.08
	WASM, Chrome	1.1	0.01
Apple A9 APL0898 (iOS 13.5)	JS, Safari	6.8	0.02
	WASM, Safari	2.4	0.02

TABLE 1. Mean PoW hash rates on different platforms. SEM = standard error of the mean, JS = JavaScript, WASM = WebAssembly.

VII. DISCUSSION

We investigated the effectiveness of PoW and CPPs as mitigation against automated threats to web applications. We introduced a general concept including recursive definition of sub-puzzles and partial server-side solution verification. The application of this concept provides high client/server asymmetry and largely reduces the possible advantage gained by an attacker through parallel processing and/or by using special hardware. We proposed and investigated a specific CPP based on partial hash inversion or hash collision and obtained some benchmark test results demonstrating the viability of the discussed approach.

To the best of our knowledge, the present study is the first one addressing the possibility of CPP protection against automated threats to web applications (with the possible exception of [23]). We also believe that the proposed concept of recursive definition of sub-puzzles and partial server-side solution verification is an original contribution introduced for the first time by this study.

The test results displayed in Table 1 were obtained with a prototype implementation of the proposed algorithm. These results show that a substantial increase in computational speed is achieved by using, e.g. Python versus JavaScript. This increase is, however, *not* large enough to render futile the proposed approach. Even with the advantage of a compiled client-side implementation of the CPP, an attacker must still pay a high price in terms of CPU resources. The same holds for the advantage gained through the usage of more powerful CPUs versus, e.g., ARM processors.

We could not compare the results from the benchmark tests to any relevant results from other studies, because, as already mentioned, we are not aware of any such studies addressing the specific problem of implementing CPPs in browser-supported languages in order to mitigate automated threats to web applications. The only exception [23] proposes a simple partial hash inversion puzzle which is easily parallelizable. The result of such parallelization is well known – a tremendous increase in hash rate to the advantage of an attacker equipped with the necessary hardware [2].

As already mentioned in the Introduction, different web applications can be protected by different CPPs. Porting a certain CPP's client-side JavaScript code to another language may increase the computational speed, but only in attacks against web applications using this particular CPP. To launch an attack against a web application protected by a different CPP, the attacker must begin from scratch. This effect may be amplified by using a multitude of small variations of the PoW algorithm within the same CPP. To make things even more difficult for the attacker, the CPP's client-side JavaScript code may be heavily obfuscated. Although methods for automated simplification of obfuscated JavaScript code do exist [32], an attacker would still have to put considerable amount of (human-expert) effort into the development of a faster implementation of the PoW algorithm, which, in the end, would not be dramatically faster than the original.

A. LIMITATIONS

The proposed approach is not equally fitting for protection against all threat events listed in the OWASP handbook [22]. For instance, it seems completely inappropriate for the mitigation of fingerprinting, footprinting, scraping, or vulnerability scanning. In these cases, each and every URL of the application is included in the attack and a CPP layer would render the whole application unacceptably slow in response. Similarly, ad fraud, skewing, cashing out, denial of inventory, snipping, scalping, expediting, spamming, and CAPTCHA defeat could hardly be thwarted by CPPs. Such attacks are based on automation of user actions which are performed (manually) very often during the normal usage of the targeted application. Hence, the deployment and activation of a CPP would result in unacceptably frequent delays.

A CPP could certainly be included as a layer of mitigation against credential cracking and credential stuffing [22]. It would, however, disrupt the user experience. Many people use their browser's password manager to store their credentials for web sites. Username and password are filled automatically into the login form as soon as it is loaded. With CPP protection in place, people would have to wait till the puzzle is solved by the browser, before proceeding further as authenticated users. Nevertheless, a CPP could still be useful as an additional layer of defense which is activated by an intrusion detection system only in case of attack against the application's login function.

We see the greatest utility of CPPs in the protection of larger web forms that take longer to fill. The time needed by a user to fill the form can simultaneously be used by the browser to solve the puzzle. Thus, the user's experience will not be affected by the protection mechanism. Short forms containing few but highly sensitive data can also be protected by CPPs. Such forms are usually submitted only once by the same user and, although their submission would take longer, users can be persuaded that this is necessary for security reasons. Examples of threat events based on automatic submission of large or sensitive user input are account creation, carding, credit card cracking, and token cracking as well as various kinds of DoS [22], including the scenario described in the Introduction. In such cases, a CPP could even be employed as the first line of defense against automated attacks.

VIII. CONCLUSION

CPPs based on recursive sub-puzzles and partial server-side verification can provide an effective layer of mitigation against various automated threats to web applications. Future research may focus on other PoW schemes that could possibly reduce further the advantage of an attacker using special hardware to solve the puzzle.

APPENDIX A PSEUDOCODE AND FLOWCHART

The following pseudocode represents the client-side algorithm used to solve the puzzle. The required input parameters are provided by the server.

Require: $\mathcal{N} > 10$ {Number of sub-puzzles}
Require: $\mathcal{C}$ = True/False {Puzzle type (True for collision)}
Require: $16 < \mathcal{B} \leq 32$ {Number of bits}
Require: $\ell > 100$ {Recursion depth}
Require: $0 \leq \mathcal{K} < 16^{48}$ {Initial key}
Require: $\mathcal{R} > 0$ {Padding increasing message size}
Require: $2^{\mathcal{B}-1} < \mathcal{H} < 2^{\mathcal{B}}$ {Target for partial inversion}
  $B \leftarrow 2^{\mathcal{B}-1}$
  $K_0 \leftarrow \mathcal{K}\|0\|0$
  for $n = 0$ **to** $\mathcal{N} - 1$ **do**
    $h^0 \leftarrow 0,\ h^1 \leftarrow 0,\ \ldots\ h^{\ell-1} \leftarrow 0$
    if $\mathcal{C}$ **then** {Puzzle type is collision}
      $I^0 \leftarrow$ False, $I^1 \leftarrow$ False, $\ldots$ $I^B \leftarrow$ False
    end if
    $X \leftarrow$ False, $i \leftarrow \ell - 1$
    while not X **do**
      $i \leftarrow i + 1$
      $M^i \leftarrow h^{i-\ell}\|h^{i-\ell+1}\|\ldots h^{i-1}\|\mathcal{R}$
      $h^i \leftarrow H_{\mathcal{B}}(K_n,\ M^i)$
      if $i \geq 2\ell$ **then**
        if $\mathcal{C}$ **then** {Puzzle type is collision}
          $X \leftarrow I^{h^i}$
          $I^{h^i} \leftarrow$ True
        else {Puzzle type is inversion}
          $X \leftarrow (h^i < \mathcal{H})$
        end if
      end if
    end while
    $S_n \leftarrow h^i$ {Sub-puzzle solution}
    $\mathbf{h}_n \leftarrow (h^{i-2\ell}, h^{i-2\ell+1}, \ldots h^{i-1})$ {PoW sequence}
    if $\mathcal{C}$ **then** {Puzzle type is collision}
      $j \leftarrow \text{IndexOf}(h^j \mid 2\ell \leq j < i,\ \ h^j = h^i)$
      $\mathbf{g}_n \leftarrow (h^{j-2\ell}, h^{j-2\ell+1}, \ldots h^{j-1})$ {2nd PoW seq.}
    else {Puzzle type is inversion}
      $\mathbf{g}_n \leftarrow$ Null
    end if
    $K_{n+1} \leftarrow \mathcal{K}\|(n+1)\|S_n$
  end for
  $\mathbf{S} \leftarrow (S_0, S_1, \ldots S_{\mathcal{N}-1})$ {Puzzle solution}
  return $\mathbf{S}, \mathbf{h}_0, \mathbf{h}_1, \ldots \mathbf{h}_{\mathcal{N}-1}, \mathbf{g}_0, \mathbf{g}_1, \ldots \mathbf{g}_{\mathcal{N}-1}$

The following pseudocode represents the server-side algorithm used for partial verification of the puzzle solution and the PoW sequences provided by the client.

Require: $\mathbf{S}$
  $(S_0, S_1, \ldots S_{\mathcal{N}-1}) \leftarrow \mathbf{S}$
  $n \leftarrow \text{Random}(0 \leq n < \mathcal{N})$
  $K_n \leftarrow \mathcal{K}\|n\|S_{n-1}$
Require: $\mathbf{h}_n, \mathbf{g}_n$
  $(h_n^0, h_n^1, \ldots h_n^{2\ell-1}) \leftarrow \mathbf{h}_n$
  $M_h \leftarrow h^{\ell}\|h^{\ell+1}\|\ldots h^{2\ell-1}\|\mathcal{R}$
  $i \leftarrow \text{Random}(\ell \leq i < 2\ell)$
  $M^i \leftarrow h^{i-\ell}\|h^{i-\ell+1}\|\ldots h^{i-1}\|\mathcal{R}$
  if $S_n \neq H_{\mathcal{B}}(K_n,\ M_h)$ **then**
    return False
  else if $h^i \neq H_{\mathcal{B}}(K_n,\ M^i)$ **then**
    return False
  else if $\mathcal{C}$ **then** {Puzzle type is collision}
    $(g_n^0, g_n^1, \ldots g_n^{2\ell-1}) \leftarrow \mathbf{g}_n$
    $M_g \leftarrow g^{\ell}\|g^{\ell+1}\|\ldots g^{2\ell-1}\|\mathcal{R}$
    $j \leftarrow \text{Random}(\ell \leq j < 2\ell)$
    $M^j \leftarrow g^{j-\ell}\|g^{j-\ell+1}\|\ldots g^{j-1}\|\mathcal{R}$
    if $S_n \neq H_{\mathcal{B}}(K_n,\ M_g)$ **then**
      return False
    else if $h^j \neq H_{\mathcal{B}}(K_n,\ M^j)$ **then**
      return False
    else
      return True
    end if
  else {Puzzle type is inversion}
    if $S_n \geq \mathcal{H}$ **then**
      return False
    else
      return True
    end if
  end if

The flowchart displayed in Figure 1 provides a schematic representation of the whole CPP.

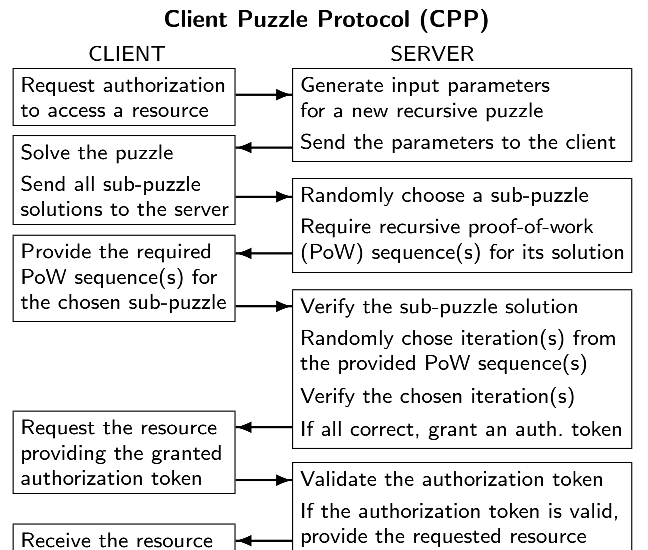

FIGURE 1. Schematic CPP representation.

ACKNOWLEDGMENT
This study was supported by the Research & Development Fund of SySS GmbH. The author is grateful to Colin Watson of OWASP and to Alex Biryukov of the University of Luxembourg for the valuable discussions.

REFERENCES
[1] S. Nakamoto. (2008) Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: https://bitcoin.org/bitcoin.pdf
[2] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, Bitcoin and cryptocurrency technologies: A comprehensive introduction. Princeton University Press, 2016.
[3] I. M. Ali, M. Caprolu, and R. D. Pietro, "Foundations, properties, and security applications of puzzles: A survey," ACM Computing Surveys (CSUR), vol. 53, no. 4, pp. 1–38, 2020.
[4] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in Annual International Cryptology Conference. Berlin: Springer, 1992, pp. 139–147.
[5] A. Back. (1997) Hashcash. [Online]. Available: http://www.hashcash.org/papers/announce.txt
[6] A. Juels and J. Brainard, "Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks," in Proceedings of NDSS '99 (Networks and Distributed Security Systems), S. Kent, Ed., 1999, pp. 151—-165.
[7] A. Back. (2002) Hashcash – a denial of service counter-measure. [Online]. Available: http://www.hashcash.org/papers/hashcash.pdf
[8] Y. I. Jerschow and M. Mauve, "Non-parallelizable and non-interactive client puzzles from modular square roots," in 2011 Sixth International Conference on Availability, Reliability and Security. IEEE, 2011, pp. 135–142.
[9] ——, "Modular square root puzzles: Design of non-parallelizable and non-interactive client puzzles," Computers & Security, vol. 35, pp. 25–36, 2013.
[10] D. Stebila, L. Kuppusamy, J. Rangasamy, C. Boyd, and J. G. Nieto, "Stronger difficulty notions for client puzzles and denial-of-service-resistant protocols," in Cryptographers' Track at the RSA Conference. Berlin: Springer, 2011, pp. 284–301.
[11] A. Juels and J. Brainard, "Cryptographic Measures against Connection Depletion Attacks," Patent US 7,197,639 B1, Mar. 27, 2007.
[12] T. Aura, P. Nikander, and J. Leiwo, "DOS-resistant authentication with client puzzles," in International workshop on security protocols. Berlin: Springer, 2000, pp. 170–177.
[13] C. Dwork, A. Goldberg, and M. Naor, "On memory-bound functions for fighting spam," in Annual International Cryptology Conference. Berlin: Springer, 2003, pp. 426–444.
[14] M. Abadi, M. Burrows, M. Manasse, and T. Wobber, "Moderately hard, memory-bound functions," ACM Transactions on Internet Technology (TOIT), vol. 5, no. 2, pp. 299–327, 2005.
[15] F. Coelho, "Exponential memory-bound functions for proof of work protocols." IACR Cryptol. ePrint Arch., vol. 2005, p. 356, 2005.
[16] A. Biryukov and D. Khovratovich, "Egalitarian computing," in Proceedings of the 25th USENIX Security Symposium, 2016, pp. 315–326. [Online]. Available: https://www.usenix.org/sites/default/files/sec16_full_proceedings.pdf
[17] ——, "Equihash: Asymmetric proof-of-work based on the generalized birthday problem," Ledger, vol. 2, pp. 1–30, 2017. [Online]. Available: https://doi.org/10.5195/ledger.2017.48
[18] A. Meneghetti, M. Sala, and D. Taufer, "A survey on PoW-based consensus," Annals of Emerging Technologies in Computing (AETiC), Print ISSN, pp. 2516–0281, 2020.
[19] D. Dean and A. Stubblefield, "Using client puzzles to protect TLS," in 10th USENIX Security Symposium (USENIX Security 01). Washington, D.C.: USENIX Association, 2001. [Online]. Available: https://www.usenix.org/conference/10th-usenix-security-symposium/using-client-puzzles-protect-tls
[20] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson, Host Identity Protocol, Request for Comments (RFC) 5201. Internet Engineering Task Force, 2008. [Online]. Available: https://tools.ietf.org/html/rfc5201
[21] X. Wang and M. K. Reiter, "Mitigating bandwidth-exhaustion attacks using congestion puzzles," in Proceedings of the 11th ACM conference on Computer and communications security, 2004, pp. 257–267.
[22] C. Watson and T. Zaw. (2018) OWASP Automated Threat Handbook Web Applications. OWASP. [Online]. Available: https://owasp.org/www-pdf-archive/Automated-threat-handbook.pdf
[23] D. Shiell, A. R. Khakpour, R. J. Peters, and D. Andrews, "Unobtrusive and Dynamic DDoS Mitigation," Patent US 10,567,427 B2, Feb. 18, 2020.
[24] H. Krawczyk, M. Bellare, and R. Canetti, HMAC: Keyed-Hashing for Message Authentication, Request for Comments (RFC) 2104. Internet Engineering Task Force, 1997. [Online]. Available: https://tools.ietf.org/html/rfc2104
[25] National Institute of Standards and Technology (NIST), The Keyed-Hash Message Authentication Code (HMAC). Federal Information Processing Standards Publication 198-1 (FIPS PUB 198-1), 2008. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.198-1.pdf
[26] D. Eastlake and T. Hansen, US Secure Hash Algorithms, Request for Comments (RFC) 6234. Internet Engineering Task Force, 2011. [Online]. Available: https://tools.ietf.org/html/rfc6234
[27] National Institute of Standards and Technology (NIST), Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-4 (FIPS PUB 180-4), 2015. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf
[28] C. Paar and J. Pelzl, Understanding cryptography: A textbook for students and practitioners, 2nd ed. Berlin: Springer, 2010.
[29] S. Nigel P, Cryptography Made Simple. Cham: Springer, 2016.
[30] Emscripten. [Online]. Available: https://emscripten.org/
[31] V. Bostanov. (2021) CPP4WebApp: A Demonstration Software Implementation of Client Puzzle Protocols as Countermeasure against Automated Threats to Web Applications. [Online]. Available: https://github.com/vladimir-bostanov/CPP4WebApp
[32] G. Lu and S. Debray, "Automatic simplification of obfuscated JavaScript code: A semantics-based approach," in 2012 IEEE Sixth International Conference on Software Security and Reliability. IEEE, 2012, pp. 31–40.

VLADIMIR BOSTANOV was born in Sofia, Bulgaria in 1969. He received a diploma in physics in 1996 from the Sofia University St. Kliment Ohridski and then worked til 1999 as a researcher in theoretical solid state physics at the Bulgarian Academy of Sciences in Sofia, the Fritz Haber Institute of the Max Planck Society in Berlin, and the University of Augsburg, Germany. He obtained a doctoral degree in psychology in 2004 from the Eberhard Karl University of Tübingen, Germany, where he spent more than 17 years as a neuroscientist. His work was focused on the development and implementation of novel mathematical and statistical methods for analysis and quantitative assessment of event-related brain potentials (ERP). He was a winner on two of the provided datasets at the International Brain-Computer Interface (BCI) Competition 2003. Currently, Dr. Bostanov works as an IT security consultant for SySS GmbH in Tübingen, Germany. The focus of his research is on web application security.