Non-Parallelizable and Non-Interactive Client Puzzles from Modular Square Roots

Yves Igor Jerschow Martin Mauve

Institute of Computer Science, Heinrich Heine University, Düsseldorf, Germany
{jerschow, mauve}@cs.uni-duesseldorf.de

***Abstract*—Denial of Service (DoS) attacks aiming to exhaust the resources of a server by overwhelming it with bogus requests have become a serious threat. Especially protocols that rely on public key cryptography and perform expensive authentication handshakes may be an easy target. A well-known countermeasure against DoS attacks are client puzzles. The victimized server demands from the clients to commit computing resources before it processes their requests. To get service, a client must solve a cryptographic puzzle and submit the right solution. Existing client puzzle schemes have some drawbacks. They are either parallelizable, coarse-grained or can be used only interactively. In case of interactive client puzzles where the server poses the challenge an attacker might mount a counterattack on the clients by injecting fake packets containing bogus puzzle parameters. In this paper we introduce a novel scheme for client puzzles which relies on the computation of square roots modulo a prime. Modular square root puzzles are non-parallelizable, i. e., the solution cannot be obtained faster than scheduled by distributing the puzzle to multiple machines or CPU cores, and they can be employed both interactively and non-interactively. Our puzzles provide polynomial granularity and compact solution and verification functions. Benchmark results demonstrate the feasibility of our approach to mitigate DoS attacks on hosts in 1 or even 10 GBit networks. In addition, we show how to raise the efficiency of our puzzle scheme by introducing a bandwidth-based cost factor for the client.**

***Keywords*—client puzzles, Denial of Service (DoS), network protocols, authentication, computational puzzles**

I. Introduction

Denial of Service (DoS) attacks pose an increasing threat to network protocols in the Internet, but also public and enterprise local area networks may be affected. Especially protocols that perform authentication and key exchange relying on expensive public key cryptography are likely to be preferred targets, e. g., SSL / TLS, IPsec, or IEEE 802.1X (EAPOL). By flooding valid-looking requests, for example authentication handshakes, an attacker may try to overload his victim. A well-known countermeasure against resource exhaustion are *client puzzles* [1]–[3]. A server being under attack processes requests only from those clients that themselves spend resources in solving a cryptographic puzzle and submit the right solution. Puzzle verification must be cheap, while the puzzle difficulty can be tuned from easy to hard. By imposing a computational task on the client the victimized server dramatically cuts down the number of valid requests that the attacker can emit. However, benign hosts having only a single request are hardly penalized. A widely-used cost function for client puzzles is the reversal of a one-way hash function by brute force. Verifying such a puzzle involves only a single hash operation.

Client puzzles can be *interactive* or *non-interactive*. In the first case the server constructs the puzzle upon receiving a request and demands from the client to solve it before continuing with the protocol. In the latter case the client constructs the puzzle by itself, solves it and attaches the solution to its request. An important characteristic of client puzzles is *granularity*, i. e., the ability to finely adjust the puzzle difficulty to different levels. Another desirable property is *non-parallelizability*, which prevents an attacker from obtaining the solution faster than scheduled by distributing the puzzle to multiple CPU cores or to other compromised machines [4]–[6]. Existing client puzzle schemes are either parallelizable, coarse-grained or can be used only interactively. Interactive puzzles have the drawback that the packet with the puzzle parameters sent from server to client lacks authentication. A second DoS attack against the clients with faked packets pretending to come from the defending server and containing bogus puzzle parameters may thwart the clients' connection attempts. Such a counterattack becomes feasible if no address authenticity is provided by the underlying layers, e. g., if operating at the link layer. To the best of our knowledge, no puzzle scheme proposed in the literature provides all the desired properties.

In this paper we introduce a novel scheme for client puzzles based on the computation of square roots modulo a prime. *Modular square root puzzles* are non-parallelizable, can be employed both interactively and non-interactively and provide polynomial granularity. We construct the puzzle for a particular request by assigning to it a unique quadratic residue a modulo a prime. Then the client solves the puzzle by extracting the modular square root of a and sends it to the server as proof of work. Computation is performed by repeated squaring, which is assumed to be an intrinsically sequential process. Verifying the puzzle on the server side is easy—it requires a single modular squaring operation and a few hash operations. Puzzle difficulty can be tuned by selecting a larger or smaller prime modulus. We evaluate the performance of modular square root puzzles by benchmarking the verification throughput and the solution time for different levels of difficulty. The results demonstrate the feasibility of our approach to mitigate DoS attacks on hosts having 1 or even 10 GBit links. To compensate for raising verification costs

in high-speed networks we strengthen our puzzle scheme by introducing a bandwidth-based cost factor for the client.

The remainder of this paper is organized as follows. In the next section, we discuss existing approaches for DoS protection with the aid of puzzles. Section III introduces algorithms for computing modular square roots, investigates parallelization aspects, and forms the mathematical basis for our client puzzles. In Section IV we then describe how to construct, solve and verify a modular square root puzzle, which can be employed in an non-interactive or interactive manner. Section V evaluates the performance of our puzzle scheme and extends it by a bandwidth-based cost factor. Finally, we conclude the paper with a summary in Section VI.

II. Related Work

A comprehensive survey on DoS attacks and proposed defense mechanisms can be found in [7]. The authors classify four categories of defense: (1) attack prevention, (2) attack detection, (3) attack source identification, and (4) attack reaction. In [1] Juels and Brainard introduced *client puzzles* to protect servers from TCP SYN flooding attacks. This countermeasure falls into the last category and constitutes a currency-based approach where clients have to pay before getting served. Being under attack, a server distributes to its clients cryptographic puzzles in a stateless manner asking them to reverse a one-way hash function by brute force. The difficulty of the puzzle is chosen depending on the attack strength. Only after receiving a correct solution from the client the server allocates resources for the dangling TCP connection. The idea of CPU-bound client puzzles has been applied to authentication protocols in general by Aura et al. in [3]. An implementation of client puzzles to protect the TLS handshake against DoS is described in [8]. Hash-reversal puzzles can be used both interactively and non-interactively. They are simple to construct and verify but have the disadvantage of being highly parallelizable and provide only exponential granularity. To make them fine-grained Feng et al. proposed *hint-based hash reversal puzzles* [9] where the server gives the client a hint about the range within which the solution lies. Thus, the granularity becomes linear. The drawback is that hint-based puzzles can be employed only interactively.

Waters et al. suggested a client puzzle scheme based on the Diffie-Hellman key exchange where puzzle construction and distribution are outsourced to a secure entity called *bastion* [10]. The bastion periodically issues puzzles for a specific number of virtual channels that are valid during the next time slot. Puzzle construction is quite expensive since it requires a modular exponentiation, but many servers can rely on puzzles distributed by the same bastion. A client solves a puzzle by computing the discrete logarithm through brute force testing—a task that is highly parallelizable. The granularity of the puzzle is linear. On the server side, verifying a puzzle involves a table lookup and another costly modular exponentiation, which, however, is performed in advance during the previous time slot.

In [4] Tritilanunt et al. introduced a non-parallelizable client puzzle scheme based on the *subset sum problem*. The client solves the puzzle by applying Lenstra's lattice reduction algorithm LLL. However, the authors point out that the memory requirements for LLL are quite high, which results in some implementation issues. Puzzle verification is quite cheap. It takes one hash operation and about 25–100 additions. Subset sum puzzles are interactive and provide polynomial granularity. In contrast, our puzzle scheme can be also employed non-interactively, has a small memory footprint, and is easy to implement.

Non-parallelizable puzzles based on repeated squaring are well-known in timed-release cryptography. In [11] Rivest et al. introduced interactive *time-lock puzzles* to encrypt messages that can be decrypted by others only after a pre-determined amount of time has passed. Like the RSA cryptosystem time-lock puzzles rely on the intractability of factoring large integers. Constructing a time-lock puzzle requires the server to perform an expensive modular exponentiation. In detail, to encrypt a message m for a period of T seconds Alice

- generates the RSA modulus $n = p\,q$ and computes $\varphi(n) = (p-1)\,(q-1)$.
- determines the number of squaring operations modulo n per second, denoted by S, that can be performed by the solver Bob, and computes $t = T \cdot S$.
- encrypts m with a symmetric cipher using the key K.
- picks a random a, $1 < a < n$, and encrypts K as

$$C_K = K + a^{2^t} mod\ n. \tag{1}$$

  To make the exponentiation efficient, Alice reduces the exponent modulo $\varphi(n)$ by computing

$$r = 2^t\ mod\ \varphi(n) \tag{2}$$

  and obtains $a^{2^t} mod\ n$ from $a^r\ mod\ n$.
- outputs the time-lock puzzle (n, a, t, C_K).

To reveal K from C_K, Bob needs to compute $a^{2^t} mod\ n$ and in contrast to Alice cannot take the shortcut via $\varphi(n)$, since determining $\varphi(n)$ is provably as hard as factoring n. Instead, Bob must do the computation step by step by repeatedly performing modular squarings—altogether t times, which is a non-parallelizable task and takes T seconds.

Seeking for a non-parallelizable (but still interactive) client puzzle scheme Karame and Čapkun adapted Rivest's puzzle by employing an RSA key pair with small private exponent to reduce the costs for puzzle verification [6]. The server must still perform a modular exponentiation but the number of multiplications is decreased by some factor, e. g., factor 12.8 for a 1024-bit modulus resulting in 120 modular multiplications instead of 1536. We find that these verification costs are nevertheless too high to provide a viable DoS protection for high-speed links. In contrast, verifying our modular square root puzzle takes only a single modular squaring operation.

With the discussed RSA based puzzle schemes we share the idea of a non-parallelizable solution function that relies on modular exponentiation. Apart from that, our approach is different and does not use any trapdoor information.

Further client puzzle architectures are, e. g., [12]–[14]. Puzzle-based DoS defense mechanisms can also rely on other payment schemes than CPU cycles, for example on memory [15]–[17], bandwidth [18], [19], or human interaction [20]. Besides DoS protection various other applications for computational puzzles have been proposed, e. g., mitigating spam [2], [21], uncheatable benchmarks [22], a zero-knowledge protocol for timed-release encryption and signatures [23], a timed commitment scheme for contract signing [24], or offline submission of documents [25].

III. Modular Square Roots

A. Extracting Square Roots Modulo a Prime

Let p be an odd prime and $a \in \mathbb{Z}_p^*$ an integer, i. e., $1 \leq a \leq p-1$. The solution of the congruence $x^2 \equiv a \,(mod\, p)$ is called a *square root modulo* p. There exist either two solutions x and $-x$ or no solution. In the first case, a is named a *quadratic residue*, and in the latter case a *quadratic non-residue* modulo p. Half of the elements in $\mathbb{Z}_p^*$ are quadratic residues and the other half are quadratic non-residues. To express whether a is a quadratic residue or not the *Legendre symbol* $\left(\frac{a}{p}\right)$ is used. It is defined as being 1 if a is quadratic residue, -1 if a is a quadratic non-residue and 0 if operating in $\mathbb{Z}_p$ and $a = 0$. The Legendre symbol can be efficiently computed in $\mathcal{O}((log\ p)^2)$ bit operations [26], [27].

Finding a square root modulo p is quite easy for half of the primes p, namely if $p \equiv 3\ (mod\ 4)$. In this case the solution is given by

$$x = a^{(p+1)/4}\ mod\ p. \tag{3}$$

For half of the remaining primes where $p \equiv 5\ (mod\ 8)$ a less trivial, but also straightforward solution exists:

$$x = \begin{cases} a^{(p+3)/8}\ mod\ p & \text{if } a^{(p-1)/4}\ mod\ p = 1 \\ 2a(4a)^{(p-5)/8}\ mod\ p & \text{otherwise.} \end{cases} \tag{4}$$

The remaining case $p \equiv 1\ (mod\ 8)$ is the most difficult one. However, there exist two well-known algorithms [28], [29] to compute square roots modulo p for all primes p, namely the *Tonelli-Shanks method* [30], [31] (see Algorithm 1 [27]) and the *Cipolla-Lehmer method* [32], [33] (see Algorithm 2 [27]). The group-theoretic Tonelli-Shanks method has a running time of $\mathcal{O}((log\ p)^4)$ bit operations if $p-1$ contains a large power of two in its prime factorization. But for small s (see line 3) it runs in $\mathcal{O}((log\ p)^3)$ since in this case the for loop is executed only a small number of times. The Cipolla-Lehmer method is based on the theory of finite fields and works with polynomials over the field $\mathbb{Z}_p$. In contrast to the algorithm of Tonelli-Shanks its running time does not depend on the decomposition of $p-1$ and is always in $\mathcal{O}((log\ p)^3)$. Note that for primes p where s is very small the Tonelli-Shanks algorithm will outperform the Cipolla-Lehmer method, because an exponentiation in the polynomial ring $\mathbb{Z}_p[x]$ is more expensive than in $\mathbb{Z}_p$. Both algorithms have a probabilistic component, namely finding a quadratic non-residue modulo p. For the Tonelli-Shanks method this quadratic non-residue does not depend on a and can be precomputed if p is fixed. A random integer $b \in \mathbb{Z}_p$ is a quadratic non-residue with probability 0.5. In case of the Cipolla-Lehmer method we need to know a to find a suitable quadratic non-residue and the probability for succeeding with a random integer b is $0.5 - \frac{1}{2p}$ [28], which converges to 0.5 for large primes p. On average, two trials should suffice for both methods to find a quadratic non-residue. The time required for this test is negligible compared to the total computation of the square root. It is an open question whether randomization can be eliminated, although this will be possible if the extended Riemann hypothesis turns out to be true. So far modular square roots can be computed only in random polynomial time by a Las Vegas algorithm [28].

Algorithm 1 Tonelli-Shanks: square roots modulo a prime p

Input: an odd prime p and an integer $a,\ 1 \leq a \leq p-1$.
Output: the two square roots of a modulo p, provided a is a quadratic residue modulo p.

1: Compute the Legendre symbol $\left(\frac{a}{p}\right)$. **if** $\left(\frac{a}{p}\right) = -1$ **then print** *"a has no square roots modulo p"* and terminate.
2: Find a quadratic non-residue b modulo p at random, i. e., an integer $b,\ 1 \leq b \leq p-1$, with $\left(\frac{b}{p}\right) = -1$.
3: Write $p - 1 = 2^s t$, where t is odd.
4: Compute $a^{-1}\ mod\ p$ by the extended Euclidean algorithm.
5: Set $c \leftarrow b^t\ mod\ p$ and $r \leftarrow a^{(t+1)/2}\ mod\ p$.
6: **for** $i = 1$ to $s-1$ **do**
7: Compute $d = (r^2 \cdot a^{-1})^{2^{s-i-1}}\ mod\ p$.
8: **if** $d \equiv -1\ (mod\ p)$ **then** set $r \leftarrow r \cdot c\ mod\ p$.
9: Set $c \leftarrow c^2\ mod\ p$.
10: **end for**
11: **return** $(r, -r)$

Algorithm 2 Cipolla-Lehmer: square roots modulo a prime p

Input: an odd prime p and an integer $a,\ 1 \leq a \leq p-1$.
Output: the two square roots of a modulo p, provided a is a quadratic residue modulo p.

1: Compute the Legendre symbol $\left(\frac{a}{p}\right)$. **if** $\left(\frac{a}{p}\right) = -1$ **then print** *"a has no square roots modulo p"* and terminate.
2: Choose an integer $b \in \mathbb{Z}_p$ at random until $b^2 - 4a$ is a quadratic non-residue modulo p, i. e., $\left(\frac{b^2-4a}{p}\right) = -1$.
3: Let f be the polynomial $x^2 - bx + a$ in $\mathbb{Z}_p[x]$. Compute $r = x^{(p+1)/2}\ mod\ f$. (Note: r will be an integer.)
4: **return** $(r, -r)$

B. Modular Exponentiation

Extracting a modular square root requires to perform modular exponentiations. This task can be accomplished by the basic *binary exponentiation method* (commonly referred to as square-and-multiply) or a more sophisticated algorithm like

the *k-ary method* or the *sliding-window method* [27]. In case $p \equiv 3 \ (mod\ 4)$ only one modular exponentiation is needed. If $p \equiv 5 \ (mod\ 8)$ then two modular exponentiations have to be performed. Finally, if $p \equiv 1 \ (mod\ 8)$ the Tonelli-Shanks or Cipolla-Lehmer algorithm has to be applied. In the worst case, namely if s is large, the Tonelli-Shanks method carries out up to $\mathcal{O}(log\ p)$ modular exponentiations in the for loop and becomes quite inefficient. Primes $p \equiv 1 \ (mod\ 8)$ of appropriate size where the prime factorization of $p-1$ contains a large power of two can be easily found. We suggest Algorithm 3 for this purpose. In line 5 the function *IsProbablePrime()* repeatedly performs a randomized primality test, e. g., the Miller-Rabin test, to achieve a given error bound (which is less than 4^{-k} after k rounds in case of the Miller-Rabin test). Finding such a "hard" prime p with an error probability below 10^{-15} takes less than 50 msec for a 1031-bit prime (input: $l = 1024$) and less than 1 sec for a 2058-bit prime (input: $l = 2048$) on a modern 64-bit CPU.

Algorithm 3 Finding a "hard" prime for modular square roots

Input: minimal bit length l.

Output: the smallest prime p having at least l bits with $p-1 = 2^s t$ where t is odd and s in $\mathcal{O}(log\ p)$.

1: set $i \leftarrow 1$
2: **repeat**
3: $\quad p = (2^{l-1} \cdot i) + 1$
4: $\quad$ set $i \leftarrow i + 2$
5: **while not** *IsProbablePrime(p)*
6: **return** p

In the following, we thus concentrate on such "hard" primes and the Cipolla-Lehmer method, which ignores the structure of $p-1$. Here the computation consists of a single modular exponentiation $x^{(p+1)/2} \ mod\ f$, but with polynomials instead of integers. The modulus f is a polynomial of degree 2 with leading coefficient 1. How many modular multiplication / squaring operations on integers are involved in this exponentiation? First, we observe that if p is a "hard" prime the exponent $(p+1)/2$ has the form $2^{s-1} \cdot i + 1$ where i is a small integer. Only some of the most significant bits and the least significant bit are set. Hence, the computation actually reduces to an exponentiation with a power-of-two exponent, where repeated squaring—a special case of the binary exponentiation—constitutes the most efficient technique. To compute $g^y \ mod\ n$ with $y = 2^k$ it takes k modular squarings and no additional multiplications while $\lfloor log\ y \rfloor$ is the lower bound for the number of multiplications to carry out a single exponentiation in a general group. Squaring a polynomial $ax + b$ of degree 1 over the field $\mathbb{Z}_p$ requires 3 modular integer multiplications / squarings. Reducing the resulting polynomial of degree 2 modulo f, i. e., performing a polynomial division, involves 2 modular multiplications and 2 modular subtractions on integers. While modular multiplication / squaring of N-bit numbers runs in $\mathcal{O}(N^2)$ (or in $\mathcal{O}(N^{1.585})$ with a sophisticated technique like Karatsuba's algorithm), modular subtraction takes linear time, and thus is negligible. Altogether, the modular exponentiation in $\mathbb{Z}_p[x]$ takes about $5 \cdot log\ p$ modular multiplication / squaring operations on integers.

C. Non-Parallelizability

In all exponentiation algorithms the main workload accounts to repeatedly performing modular squarings. This is assumed to be an intrinsically sequential, i. e., non-parallelizable process since each next step requires the intermediate result from the previous one [11]. Parallelization of the squaring operation itself cannot achieve a significant speedup either. Each squaring requires only trivial computational resources and any non-trivial scale of parallelization inside the squaring operation would be likely penalized by communication overhead among the processors [23]. In complexity theory, the class P contains all decision problems that can be solved by a deterministic Turing machine in polynomial time. $\mathsf{NC} \subseteq \mathsf{P}$ represents the class of problems that can be efficiently solved by a parallel computer. However, it is still an open question whether modular exponentiation is P-complete, i. e., not in NC [34], [35]. Likewise, it is unknown if factoring is really not in P.

We now want to point out those parts of modular square root computation that are parallelizable. If applying the basic binary exponentiation method the $\frac{1}{2} \cdot log\ p$ multiply steps can be performed in parallel to the $log\ p$ squaring steps. Thus, only $log\ p$ sequential modular squaring operations can be accounted for when extracting a square root modulo $p \equiv 3 \ (mod\ 4)$. The same applies to the case $p \equiv 5 \ (mod\ 8)$ where two modular exponentiations are performed (see Equation 4). Instead of evaluating $a^{(p-1)/4} \ mod\ p$ first and then deciding on which will be the second exponentiation, one could carry out all three modular exponentiations in parallel and then determine the correct square root instantly by checking the result of $a^{(p-1)/4} \ mod\ p$. When dealing with "hard" primes $p \equiv 1 \ (mod\ 8)$ parallelization is also possible to some degree. We can do the 3 modular multiplications / squarings to square the polynomial simultaneously. Afterwards the 2 modular multiplications for polynomial division can be also performed in parallel. This results in about $2 \cdot log\ p$ sequential modular multiplications to compute a square root modulo a "hard" prime $p \equiv 1 \ (mod\ 8)$ and takes more than twice as long as for other primes, since multiplying is somewhat slower than squaring [36]. Thus we have found a way to increase the time for square root extraction by more than factor 2, which cannot be diminished by raising the number of available processors.

IV. Client Puzzles from Modular Square Roots

A. Constructing and Solving a Non-Interactive Puzzle

The benign host A having a request (e. g., an authentication handshake) to host B that is under a DoS attack constructs for its request a unique puzzle. We suppose that both parties share a list $L = \{p_1, ..., p_j\}$ of "hard" primes $p \equiv 1 \ (mod\ 8)$ with different bit lengths which have been generated once and henceforth can be used by all hosts an unlimited number of times. The puzzle must be bound to A's request message m. Depending on the layer the protocol is operating at m may be

an Ethernet frame, an IP datagram or a TCP/UDP segment. First, host A selects from the list L a prime p of appropriate bit length n and applies a cryptographic hash function H with digest length k on m recursively $c = \lceil \frac{n}{k} \rceil$ times to produce the $(n-1)$-bit digest

$$d = First_{n-1}(H(m) \mid\mid H(H(m)) \mid\mid ... \mid\mid H^c(m)). \quad (5)$$

Here $||$ denotes the concatenation of two bit strings and $First_i$ extracts the first i bits from a bit string. Next host A considers d as a $(n-1)$-bit number and computes the Legendre symbol $\left(\frac{d}{p}\right)$ to check whether d is a quadratic residue modulo p. If it turns out to be a a quadratic non-residue, d is decremented by one until the quadratic residue a is found:

```
set a ← d
while (a/p) = −1 do
   set a ← a − 1
end while
return a
```

Since half of the elements in $\mathbb{Z}_p^*$ are quadratic residues, a few trials will usually suffice. Now, a unique quadratic residue a has been assigned to A's request. The puzzle to solve is the computation of the square root of a modulo p by applying the Cipolla-Lehmer method, which takes about $2 \cdot log\ p$ sequential modular multiplications. Without parallelization, about $5 \cdot log\ p$ modular multiplications / squarings have to be performed. Having extracted the square root x, host A attaches this n-bit number to its request and sends it to host B. The other square root $-x$ is of no importance for the protocol. There is no need to transmit the prime p. Host A can simply indicate the modulus by stating its position in the list L. Usually, all primes in the list will differ in size so that the corresponding prime may even be deduced from the size of x.

Due to the non-interactive puzzle construction an attacker might compute the puzzle solutions in advance. If precomputation is an issue, it can be mitigated by concatenating the message m with an unpredictable, periodically changing number pior to producing the digest d. Lottery results [2] or stock market prices are possible sources of randomness which are easily accessible to both parties A and B. In this case host B will accept only requests bearing an up-to-date random number.

B. Puzzle Verification

The victimized host B verifies the puzzle solution x prior to allocating resources and processing host A's request, which may require to perform a public or even private key operation or an expensive database lookup. Puzzle verification is quite cheap—besides a few hash operations (c times, depends on the hash size and the length of the prime) to compute the digest d from the request only a single modular squaring operation $x^2\ mod\ p$ has to be carried out. Host B does not necessarily need to evaluate the Legendre symbol in the while loop above to find the quadratic residue a. With probability 0.5 we have $a = d$, with probability 0.25 we have $a = d - 1$ and so on. Thus, if $d - (x^2\ mod\ p) < \delta$ where δ is a small constant, e. g., $\delta = 20$, the verification can be considered as successful, otherwise A's request is dropped. The puzzle solver A cannot take any advantage of extracting the modular square root from $a' = a - \beta$ instead of from a if β is bounded by the small constant δ. Even if host A cheats in this manner for some reason, host B can be certain that A has indeed computed a modular square root specially for its request m.

Host B's decision whether to allocate resources for processing A's request or not can, of course, also depend on the puzzle difficulty (that is, on the size of the chosen prime) and on the strength of the ongoing DoS attack. The rate of accepted requests with correct puzzle solutions shall not exceed host B's processing capacity, i. e., the rate at which B can actually complete these requests. Being rejected, host A may then retry by taking a larger prime from the list L and solving a more difficult puzzle.

C. Interactive Client Puzzles

Our modular square root puzzles can be also employed in an interactive way, where the victimized server (host B) issues a challenge to the client (host A), as is the case with client puzzles proposed by Juels and Brainard [1] and reworked by Aura et al. [3]. In the interactive setting the prime modulus p and the quadratic residue a are dictated by the server. This can be done in a stateless manner by hashing the client's request along with a secret number to produce the digest d and sending d back to the client, which derives from it the quadratic residue a for the puzzle. Thus, the server needs to store only the secret number and the prime which are reused across all clients. The advantages of interactive client puzzles are the prevention of precomputation and the precise choice of the puzzle's level of difficulty since it is prescribed by the defending server. However, a major drawback of interactive client puzzles that we have already indicated in the introduction is the lack of authentication for the packet containing the puzzle parameters, which the server sends to the client. A second DoS attack against the clients with faked packets bearing the server's sender address and containing bogus puzzle parameters may thwart the clients' connection attempts. The feasibility of such a counterattack depends on the network environment and the attacker's location. Forging the sender address and eavesdropping on the traffic is especially easy in wired and wireless LANs while it is more difficult in the Internet. Hence, only in environments where counterattacks on the clients are very unlikely, our square root puzzles should be used in the interactive manner.

D. Puzzle Granularity and Public Auditability

The ability to finely adjust the puzzle difficulty to different levels represents an important criterion for the practical applicability of a puzzle. Solving a modular square root puzzle with an N-bit prime takes $\mathcal{O}(N^3)$ time while the verification runs in $\mathcal{O}(N^2)$. Thus, having polynomial granularity, our puzzle is quite fine-grained. In contrast, a non-interactive puzzle scheme based on hash-reversal has exponential granularity

and is highly parallelizable. Since a third party can efficiently verify the solution of the square root puzzle without access to any trapdoor information, its cost-function is called *publicly auditable* [2]. Time-lock [11] and Diffie-Hellman based [10] puzzles are, by contrast, not publicly auditable.

V. Evaluation and Protocol Enhancements

In this section we evaluate the performance of our puzzle scheme and enhance it by introducing a bandwidth-based cost factor for the client.

A. Puzzle Benchmark

For "hard" primes of different size ranging from 264 to 8206 bits we measure the number of modular square root puzzles that an off-the-shelf Intel Core 2 Quad Q9400 2.66 GHz CPU can verify per second and the time it takes to solve a puzzle. Table I presents our benchmark results averaged over 10 runs. In all test series the coefficient of variation was below 1.5%. For the large-integer arithmetic we employ the well-known open source library *GMP* from GNU [36], which claims to be faster than any other bignum library by using state-of-the-art algorithms with highly optimized assembly code. Modular square root extraction is done using the Cipolla-Lehmer method, where the exponentiation in $\mathbb{Z}_p[x]$ constitutes the main workload. In our measurements we take only the time to perform $2 \cdot log\ p$ sequential modular multiplications into account, since the remaining $3 \cdot log\ p$ modular multiplications / squarings can be computed in parallel by a well-versed attacker (see Section III-C). To accelerate the repeated modular multiplications we make use of Montgomery reduction instead of performing the classical reduction by dividing. This results in a speed-up by a factor of 1.2–2.0, especially for small moduli in the order of 264–2058 bits. In our benchmark we perform all computations using a single CPU core. For full parallelization of a puzzle an attacker would employ three CPU cores while the defending host can verify as many puzzles in parallel as CPU cores are available. Solving a puzzle on a benign host that uses only a single CPU core actually takes about two and a half times longer than stated in Table I.

TABLE I
Benchmark: Verifying and Solving Modular Square Root Puzzles on Intel Core 2 Quad Q9400 2.66 GHz.

bit length	*modular squarings / sec*		*modular square root: time in msec*	
	32-bit	64-bit	32-bit	64-bit
264	1 377 000	2 597 000	0.238	0.091
520	593500	1354000	1.35	0.411
776	329 400	698 300	4.15	1.10
1031	201 300	549 400	9.01	2.42
1547	102 500	337 400	27.7	7.09
2058	62 810	199 100	62.9	15.7
3084	33 030	117 100	196	48.1
4106	20 530	71 630	429	109
6155	10 620	39 250	1350	340
8206	6810	24 430	3020	763

Evaluating the benchmark results, we first observe that a 64-bit implementation outperforms its 32-bit counterpart by a factor of up to 3.7 in verifying and up to 4.0 in solving a puzzle. Since almost all desktop CPUs manufactured during the last five years are 64-bit capable and 64-bit operating systems are widely available, we consider the 64-bit results as reference values. Secondly, the speed gap between the verifier and the solver constitutes factor 236 for a 264-bit puzzle and increases up to factor 18 640 for a 8206-bit puzzle. Now the main question to pose is whether the verification throughput of modular square root puzzles is high enough to cope with a DoS flooding attack of bogus puzzle solutions mounted at full link speed. Of course, the size of a valid-looking request containing a puzzle solution plays a role. Before we can definitely answer this question with "yes" for networks with 100 MBit, 1 GBit, and even 10 GBit links, we extend the puzzle protocol by a small bandwidth-based cost factor for the client.

The victimized host demands that valid puzzle solution packets must be padded with zeros to have full MTU (Maximum Transmit Unit) size. In the Internet, the MTU usually is 1500 bytes (in Gigabit Ethernet even up to 9000 bytes). Hence, besides solving a puzzle the client must additionally pay with bandwidth. Using bandwidth as a currency for DoS protection is a known approach in the literature [18], [19]. Now, dealing with 1500 byte packets, the victimized host will receive up to 8300 (100 MBit link), 83 000 (1 GBit link) or 830 000 (10 GBit link) valid-looking puzzle solutions per second. We note that it will perfectly cope with 8206-bit puzzles on a 100 MBit link, with 3084-bit puzzles on a 1 GBit link and with 520-bit puzzles on a 10 GBit link assuming a single CPU core engaged in puzzle verification. The time to compute the digest d must also be taken into account. But only the meaningful part of the request and not the whole packet needs to be hashed, while cryptographic hash functions like MD5 or SHA-1 process about 2.8–3.6 GBit of data per second on our test machine. Furthermore it is conceivable to produce the $(n-1)$-bit digest d by applying a very fast pseudorandom number generator to $H(m)$ instead of executing the hash function c times. On the opposite side it takes an attacker 763 msec to solve a 8206-bit puzzle, 48.1 msec to solve a 3084-bit puzzle, and 0.411 msec to solve a 520-bit puzzle, respectively, assuming full parallelization. Though for modular square root puzzles the level of difficulty cannot be chosen arbitrarily high without rendering the verification too expensive, we are convinced that the presented solution times in the order of 0.1 to 1000 msec are fully viable for DoS prevention in practice. Solution times much greater than 1 second are possible with hash-reversal puzzles, but for benign clients such long delays seem to be hardly reasonable.

Fast modular exponentiation has been also successfully implemented in hardware, especially on FPGAs [37], [38], and for modern GPUs [39], [40], which are very competitive. A few years ago FPGAs outperformed ordinary software implementations, but a current comparison [39] shows that nowadays FPGAs are about as fast as software implementations on up-to-date CPUs. A GPU implementation pays off when performing a large number of modular exponentiations simultaneously. However, this comes at the expense of high la-

tency. A speed-up of up to 4 times compared to a modern CPU has been reported in [40]. Though an experienced attacker can benefit from such hardware acceleration, his advantage over a regular solver running a software implementation is bounded by a small factor. In general, this is not an issue for the client puzzle protocol.

B. Increasing the Bandwidth-Based Payment

Besides prescribing that puzzle solution packets must be padded to have full MTU size we may go a step further and increase the bandwidth-based payment requested from the client. The victimized host can demand multiple copies of the puzzle solution packet prior to processing the associated request. This enables us to employ more complex puzzles in high-speed networks and thus to strengthen the DoS protection. For example, by prescribing that clients must send four copies of their puzzle solution packet we can cut down on the number of valid-looking puzzle solutions received per second by factor four and verify even 8206-bit puzzles on a 1 GBit link. Sending multiple copies of the puzzle solution packet is feasible for all clients regardless of their link speed, while DoS protection schemes based solely on bandwidth payment penalize clients behind slow links. To implement this protocol extension, the victimized host must maintain a packet counter for each client. An appropriate data structure for this purpose is a hash map with the client's address as the key and the pair $<$*packet counter, timestamp*$>$ as the value. Elements with old timestamps must be purged periodically from the hash map. Storage overhead for maintaining the counters is fairly low: Assuming 10 bytes per client, a 1 GBit link with 83 000 packets / sec, and a maximum lifetime of 5 sec for each entry, the size of the hash map will be about 10 MB (depending on implementation and pointer size).

VI. Conclusion

In this paper we have introduced a novel client puzzle scheme based on modular square roots as a countermeasure against DoS attacks. A modular square root puzzle is non-parallelizable, i. e., the solution cannot be obtained faster than scheduled by distributing the puzzle to multiple machines or CPU cores. Our puzzles can be employed non-interactively, which prevents counterattacks on the client mounted by injecting packets with fake puzzle parameters. Providing polynomial granularity and compact solution and verification functions, modular square root puzzles can be easily implemented to safeguard network protocols, especially those performing expensive public key authentication, against DoS. We have shown how to raise the efficiency of our puzzle scheme by introducing a bandwidth-based cost factor for the client and demonstrated its feasibility in 1 and 10 Gigabit networks through benchmarking.

References

[1] A. Juels and J. G. Brainard, "Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks," in *NDSS '99: Proceedings of the Network and Distributed System Security Symposium*, Feb. 1999.

[2] A. Back, "Hashcash - A Denial of Service Counter-Measure," Aug. 2002, http://www.hashcash.org/papers/hashcash.pdf.

[3] T. Aura, P. Nikander, and J. Leiwo, "DOS-Resistant Authentication with Client Puzzles," in *Revised Papers from the 8th International Workshop on Security Protocols*, Apr. 2001, pp. 170–177.

[4] S. Tritilanunt, C. Boyd, E. Foo, and J. M. G. Nieto, "Toward Non-Parallelizable Client Puzzles," in *CANS 2007: Proceedings of the 6th International Conference on Cryptology & Network Security*, Dec. 2007, pp. 247–264.

[5] P. Schaller, S. Čapkun, and D. Basin, "BAP: Broadcast Authentication Using Cryptographic Puzzles," in *ACNS '07: Proceedings of the 5th International Conference on Applied Cryptography and Network Security*, Jun. 2007, pp. 401–419.

[6] G. O. Karame and S. Čapkun, "Low-Cost Client Puzzles based on Modular Exponentiation," in *ESORICS 2010: Proceedings of the 15th European Symposium on Research in Computer Security*, Sep. 2010, pp. 679–697.

[7] T. Peng, C. Leckie, and K. Ramamohanarao, "Survey of Network-Based Defense Mechanisms Countering the DoS and DDoS Problems," *ACM Computing Surveys*, vol. 39, no. 1, p. 3, 2007.

[8] D. Dean and A. Stubblefield, "Using Client Puzzles to Protect TLS," in *SSYM'01: Proceedings of the 10th USENIX Security Symposium*, Aug. 2001.

[9] W.-c. Feng, E. Kaiser, W.-c. Feng, and A. Luu, "The Design and Implementation of Network Puzzles," in *INFOCOM 2005: Proceedings of the 24th IEEE Conference on Computer Communications*, Mar. 2005, pp. 2372–2382.

[10] B. Waters, A. Juels, J. A. Halderman, and E. W. Felten, "New Client Puzzle Outsourcing Techniques for DoS Resistance," in *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, Oct. 2004, pp. 246–256.

[11] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release Crypto," Massachusetts Institute of Technology, Cambridge, MA, USA, Tech. Rep., 1996.

[12] X. Wang and M. K. Reiter, "Defending Against Denial-of-Service Attacks with Puzzle Auctions," in *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003, pp. 78–92.

[13] X. Wang and M. K. Reiter, "Mitigating Bandwidth-Exhaustion Attacks using Congestion Puzzles," in *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, Oct. 2004, pp. 257–267.

[14] H. Hlavacs, W. N. Gansterer, H. Schabauer, J. Zottl, M. Petraschek, T. Hoeher, and O. Jung, "Enhancing ZRTP by using Computational Puzzles," *Journal of Universal Computer Science*, vol. 14, no. 5, pp. 693–716, 2008.

[15] M. Abadi, M. Burrows, M. Manasse, and T. Wobber, "Moderately Hard, Memory-bound Functions," *ACM Transactions on Internet Technology*, vol. 5, pp. 299–327, May 2005.

[16] C. Dwork, A. Goldberg, and M. Naor, "On Memory-Bound Functions for Fighting Spam," in *CRYPTO '03: Proceedings of the 23th Annual International Cryptology Conference on Advances in Cryptology*, Aug. 2003, pp. 426–444.

[17] S. Doshi, F. Monrose, and A. D. Rubin, "Efficient Memory Bound Puzzles Using Pattern Databases," in *ACNS 2006: Proceedings of the 4th International Conference on Applied Cryptography and Network Security*, Jun. 2006, pp. 98–113.

[18] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker, "DDoS Defense by Offense," in *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, Sep. 2006, pp. 303–314.

[19] Y. I. Jerschow, B. Scheuermann, and M. Mauve, "Counter-Flooding: DoS Protection for Public Key Handshakes in LANs," in *ICNS 2009: Proceedings of the 5th International Conference on Networking and Services*, Apr. 2009, pp. 376–382.

[20] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford, "CAPTCHA: Using Hard AI Problems For Security," in *EUROCRYPT '03: Proceedings of the 22nd International Conference on Theory and Applications of Cryptographic Techniques*, May 2003, pp. 294–311.

[21] C. Dwork and M. Naor, "Pricing via Processing or Combatting Junk Mail," in *CRYPTO '92: Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, Aug. 1992, pp. 139–147.

[22] J.-Y. Cai, R. J. Lipton, R. Sedgewickand, and A. C.-C. Yao, "Towards

uncheatable benchmarks," in *Proceedings of the 8th Annual Structure in Complexity Theory Conference*, May 1993, pp. 2–11.
[23] W. Mao, "Timed-Release Cryptography," in *SAC 2001: Proceedings of the 8th Annual International Workshop on Selected Areas in Cryptography*, Aug. 2001, pp. 342–357.
[24] D. Boneh and M. Naor, "Timed Commitments," in *CRYPTO '00: Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology*, Aug. 2000, pp. 236–254.
[25] Y. I. Jerschow and M. Mauve, "Offline Submission with RSA Time-Lock Puzzles," in *CIT 2010: Proceedings of the 10th IEEE International Conference on Computer and Information Technology*, Jun. 2010, pp. 1058–1064.
[26] H. Cohen, *A Course in Computational Algebraic Number Theory*. Springer, 1996.
[27] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.
[28] E. Bach and J. Shallit, *Algorithmic Number Theory, Volume I: Efficient Algorithms*. MIT Press, 1996.
[29] N. Nishihara, R. Harasawa, Y. Sueyoshi, and A. Kudo, "A remark on the computation of cube roots in finite fields," Cryptology ePrint Archive, Report 2009/457, 2009, http://eprint.iacr.org/2009/457.
[30] A. Tonelli, "Bemerkung über die Auflösung quadratischer Congruenzen," *Göttinger Nachrichten*, pp. 344–346, 1891.
[31] D. Shanks, "Five number-theoretic algorithms," in *Proceedings of the 2nd Manitoba Conference on Numerical Mathematics*, 1972, pp. 51–70.
[32] M. Cipolla, "Un metodo per la risolutione della congruenza di secondo grado," *Rendiconto dell'Accademia Scienze Fisiche e Matematiche*, vol. 9, no. 3, pp. 154–163, 1903.
[33] D. H. Lehmer, "Computer technology applied to the theory of numbers," *Studies in Number Theory, Prentice Hall, Englewood Cliffs, NJ*, pp. 117–151, 1969.
[34] L. Adleman and K. Kompella, "Using Smoothness to Achieve Parallelism," in *STOC '88: Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, 1988, pp. 528–538.
[35] J. P. Sorenson, "A Sublinear-Time Parallel Algorithm for Integer Modular Exponentiation," in *Proceedings of the Conference on the Mathematics of Public-Key Cryptography*, Jun. 1999, pp. 528–538.
[36] "GMP: GNU Multiple Precision Arithmetic Library," http://gmplib.org.
[37] M. M. Ciaran McIvor, J. McCanny, A. Daly, and W. Marnane, "Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures," in *Proceedings of the 37th Asilomar Conference on Signals, Systems, and Computers*, Nov. 2003, pp. 379–384.
[38] D. Suzuki, "How to Maximize the Potential of FPGA Resources for Modular Exponentiation," in *CHES '07: Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems*, Sep. 2007, pp. 272–288.
[39] R. Szerwinski and T. Güneysu, "Exploiting the Power of GPUs for Asymmetric Cryptography," in *CHES '08: Proceedings of the 10th International Workshop on Cryptographic Hardware and Embedded Systems*, Aug. 2008, pp. 79–99.
[40] O. Harrison and J. Waldron, "Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware," in *AFRICACRYPT '09: Proceedings of the 2nd International Conference on Cryptology in Africa*, Jun. 2009, pp. 350–367.