Efficient Client Puzzles based on Repeated-Squaring

Ghassan O. Karame
ETH Zürich, Switzerland
karameg@inf.ethz.ch

Srdjan Čapkun
ETH Zürich, Switzerland
capkuns@inf.ethz.ch

***Abstract*—In this paper, we propose a new, non-parallelizable verification-efficient client puzzle. Our puzzle is based on repeated-squaring and enables efficient verification of the puzzle solution that is reported by the client (prover). Client puzzles based on repeated-squaring were first proposed by Rivest et al. in [1] and constitute one of the first examples of non-parallelizable puzzles. The main drawback of these puzzles was their high verification overhead. In this work, we show how this overhead can be significantly reduced by transferring the puzzle verification burden to the prover that executes the puzzle. Given a 1024-bit modulus, the improvement gain in the verification overhead of our puzzle when compared to the original repeated-squaring puzzle is almost 50 times. We achieve this by embedding a secret – only known to the verifier – within the Euler trapdoor function that is used in repeated-squaring puzzles. We provide a security proof for this construction. We further show how our puzzle can be integrated in a number of protocols, including those used for efficient protection against DoS attacks and for the remote verification of the computing performance of devices. We validate the performance of our puzzle on a large number of PlanetLab nodes.**

I. Introduction

In this work, we address the problem of the efficient verification of computational client-puzzles that are based on repeated squaring. Client puzzles found their application in a number of domains, but their main applications concern their use for the protection against Denial-of-Service (DoS) attacks [2], [1], [3] and for the verification of the computing performance [4], [5]. In the context of DoS attacks, client puzzles are used by servers that require their clients to solve a computational puzzle before attending to a request. In the context of the verification of computing performance, puzzles are used as "uncheatable" benchmarks that verify the processing performance of a device or of a set of devices to prevent false performance claims; reports of false claims of processor benchmarks have been recently reported [6], [7], [8].

A number of computational (CPU-bound) puzzles have been proposed [3] but these proposals are either efficient and parallelizable [2], [9] or non-parallelizable and inefficient (typically in result verification) [1]. Puzzles have equally been proposed as "uncheatable" benchmarks [4], [5]; these benchmarks are, however, parallelizable. Furthermore, existing computational puzzles are often considered impractical because the time it takes to solve them can vary across computing platforms [10].

To be useful in practice, client puzzles have to satisfy several criteria: namely, they need to be secure, inexpensive to construct and verify, and in many applications should be non-parallelizable. Non-parallelizability of puzzles is an especially important property since clients can involve other processors at their disposal to inflate their problem-solving performance claim. This property is equally important in scenarios where a verifier (server) needs to verify the performance of a single processor and in protocols for DoS protection since it can significantly reduce the effectiveness of distributed DoS attacks. Furthermore, the verifier has to ensure the fairness of the entire process given the large disparity in the resources of its clients; all clients – even those running on modest machines – should spend comparable time and resources in solving the puzzle. These challenges hinder the large-scale deployment of computational puzzles in today's online applications.

In this paper, we consider this problem and we propose an efficient, yet non-parallelizable, computational puzzle that significantly improves the performance of existing CPU-bound puzzles. Our puzzle is based on repeated-squaring and uses the puzzle proposed in [1] as its basic building block, but "outsources" most of verification of the puzzle's solution to the prover; this is achieved without compromising the application goals (e.g., DoS resilience, computing performance verification) by embedding a secret – only known to the verifier – within the trapdoor exhibited by the Euler function in modular squaring. We provide a security proof for this construction. To verify the client's solution, the verifier only needs to execute a marginal number of modular multiplications ($O(\log(v))$, where v is a typically small integer, e.g., a 20-bit integer). For example, for a 1024-bit modulus N, the improvement gain in the verification of our puzzle when compared to the original repeated-squaring puzzle [1] is $\frac{\log(N)}{\log(v)} \simeq 50$ times. In the context of DoS protection and other real-time applications, this improvement is considerable; given our puzzle, an online server can verify the solution of 50 different squaring puzzles (or queries) using the same resources/time that it would originally take the server to verify the solution of a single execution of the original repeated-squaring puzzle. We validate the performance of our puzzle through experiments on a large number of PlanetLab nodes [11].

Besides proposing a new client puzzle, in this work, we make the following additional contributions. In relation to the proposed puzzle, we show how it can be integrated in protocols used for the DoS protection and for the remote verification of computing performance. We further show how puzzle-based secure verification of device computing performance can be used to enhance fairness in puzzle-based protection from DoS attacks; this scheme can be used with all non-parallelizable puzzles. Owing to its efficiency, our puzzle is equally well suited for scenarios in which low-end devices (e.g., PDAs or sensor nodes) verify the computational performance of high-end processors.

The rest of the paper is organized as follows. In Section II, we overview the related work. In Section III, we introduce our protocol and we analyze its resilience to a multitude of security threats. Section IV outlines some applications that can benefit from our proposed scheme. In Section V, we discuss further insights related to our proposals and we conclude the paper in Section VI.

II. Related Work

In what follows, we briefly overview related work in the area.

Client Puzzles: Client puzzles found their application in several application domains (e.g., prevention against DoS attacks [12], [13], protection from connection depletion attacks [14], etc.). Several computational puzzles have been proposed in the recent years [2], [1], [3]. A comprehensive survey of existing client puzzles can be found in [3]. In [1], Rivest *et al.* proposed a non-parallelizable puzzle based on repeated squaring to enable time-release cryptography. The drawback of this scheme, if used for DoS protection, is that it requires expensive computations in the puzzle verification stage, which renders it less efficient in countering DoS attacks. Another approach to building computational puzzles is HashCash [15], originally proposed as a countermeasure to email spam. However, in HashCash, an attacker can pre-compute all the solution tokens and temporarily overload the system [16]. Dean *et al.* show in [17] the applicability of CPU bound puzzles in protecting SSL against denial of service attacks. Wang *et al.* propose in [13] a scheme that enables the server to adjust the puzzle difficulty in the presence of an adversary whose computing power is unknown. Our puzzle, on the other hand, allows the server to evaluate the computing performance of its clients in order to adjust the puzzle difficulty accordingly (Section IV-B).

Memory-bound puzzles were proposed in [18], [16] to overcome the limitations of existing computational (CPU-bound) puzzles. These puzzles leverage on the low disparity in memory access times and therefore converge faster than the corresponding computational puzzles. However, memory-bound puzzles cannot entirely substitute their CPU-bound counterpart e.g., in applications where the client's memory is limited (e.g., PDA devices) or in the evaluation of the computing performance of devices, etc..

Several other contributions address the problem of secure outsourcing of computations to untrusted servers (e.g., [19], [20]). In [21], Jakobsson *et al.* propose a scheme that enables secure outsourcing of a batch of signatures to a remote server. Clarke *et al.* present protocols for speeding up exponentiation using untrusted servers in [22]. In [23], Hohenberger *et al.* describe a scheme to outsource cryptographic computations (i.e., modular exponentiation) where the verifier can use two untrusted exponentiation programs to assist him in the computations. In this work, we show that efficient outsourcing of exponentiation programs can be achieved using a single untrusted remote program.

Uncheatable Benchmarks: The notion of secure and "uncheatable" benchmarks to evaluate a machine's computational performance was first introduced in [4] and [5]. In [4], Cai *et al.* argue for the need of secure benchmarks and briefly introduce a secure benchmark based on repeated-squaring. The authors further propose in [5] a set of benchmark candidates that are resistant to tampering by untrusted hosts. Namely, they rely on complexity theory to strengthen FFT, Gaussian Elimination and Matrix Multiplication-based benchmarks. However, the underlying algorithms that implement FFT, Gaussian Elimination and Matrix Multiplication can be easily parallelized as explained in [24], [25], [26], respectively. For instance, by parallelizing a Gaussian Elimination-based benchmark (e.g., Linpack [27]), the gain in computational performance increases almost linearly with the number of processors used.

Obfuscation of Executable Code: Another solution to create secure puzzles would be for the verifier to send an obfuscated executable code [28] whose solution is already pre-computed. Assuming perfect obfuscation, this method might in theory prevent an untrusted prover from disassembling and understanding the code, and therefore might harden the parallelization and/or the tampering with the code. However, existing code obfuscation techniques can only result in modest, best-effort efficacy nowadays [28], [29]. Furthermore, these techniques come at the expense of additional complexity and overhead at the puzzle construction phase.

III. Efficient Repeated-Squaring Puzzle

In this section, we introduce our system and attacker model and we present our puzzle.

A. System and Attacker Model

We consider the following model. A verifier (typically equipped with a device that has limited computation power) requests that a prover solves a computational puzzle in a certain amount of time.

For that purpose, the verifier requires that the prover runs a software on its machine (e.g., a CPU-bound puzzle or a benchmark code) for a specific amount of time. In some application scenarios, we will need to assume that the verifier and the prover can exchange authenticated messages over the communication channel (e.g., by signing their messages using shared keys). We assume, however, that the verifier does not have access to the prover's machine and thus cannot check the prover's environment; this includes the prover's CPU usage, the number of processors at the disposal of the prover, the connections established from the prover's machine, etc..

An untrusted prover constitutes the core of our attacker model. We assume that a prover possesses considerable technical skills by which it can efficiently analyze, decompile and/or modify executable code as necessary. More specifically, an untrusted prover has knowledge of both of the algorithm used for the computation and of the measures used by the verifier to prevent potential tampering with the evaluation process. However, we assume that untrusted provers are computationally bounded.

We further assume that untrusted provers are motivated to inflate their puzzle solving performance (i.e., untrusted provers have incentives to solve the puzzle in a faster time than what they can genuinely perform) and we do not address performance deflation attacks. In Section IV, we show that this is a reasonable assumption in a number of applications. Untrusted provers might attempt to inflate their puzzle-solving performance as follows: *(i)* untrusted provers might simply not run the puzzle sent by the verifier and return incorrect results ahead of time, *(ii)* untrusted provers might involve other machines at their disposal to speed up the execution time of the puzzle and *(iii)* untrusted provers might try to find a shortcut to compute the correct solution of the puzzle ahead of time.

B. Efficient Repeated-Squaring Puzzle

Our puzzle is inspired by the pseudo-random generator proposed by Blum *et al.* in [30] and its time-lock puzzle variant proposed by Rivest *et al.* in [1]. The main intuition behind [1] is that the prover is required to compute the outcome of a repeated-squaring operation, $X^{a^t} \bmod N$, given a generator X, a base exponent a, a large integer t and an appropriate modulus N. It is known that this computation can be verified through the shortcut offered by Euler's function:

$$X^{a^t} \; mod \; N = X^{a^t \; mod \; \phi(N)} \; mod \; N$$

However, this verification requires $O(\log(N))$ modular multiplications – which might be expensive for a verifier with a modest computational capability [3] or for the one serving a large number of clients. In what follows, we describe our puzzle, which makes this verification significantly more efficient; in our puzzle, the verification of the puzzle's solution only requires $O(\log(v))$ modular multiplications, where v is typically a small e.g., 20-bit integer.

Our puzzle and the related protocol are shown on Figure 1. Our puzzle is composed of three phases: a *pre-computation* phase, performed only once by the verifier, a *puzzle construction* phase and a *solution verification* phase.

Prior to starting our protocol, we assume that the verifier pre-computes $N = p \cdot q$ and $\phi(N) = (p-1)(q-1)$. The verifier equally picks a generator X, a large integer M and calculates $m \equiv X^{M \; mod \; \phi(N)} \bmod N$. M acts as a secret that is only known to the verifier; we show later that by leveraging on the secrecy of M, the secure verification of repeated-squaring can be efficiently achieved. The pre-computation of m is performed only once and requires $O(\log(N))$ modular multiplications.

The verifier picks two integers a, v and a large exponent t (typically $t > 1,000,000$). We point that v is a small integer (e.g., a 20-bit integer); within $\sqrt{2^L}$ (L refers to the size of v in bits) different consecutive puzzles, the values of v must be *distinct* (see Section III-D). The verifier then asks the prover to compute $Y_1 \equiv X^{a^t} \bmod N$. We show later that any choice of a does not deteriorate the security of our scheme. Note that by requiring that the prover performs repeated-squaring, the time required to generate the optimal multiplication sequence using addition-chains [31] is significantly reduced since the optimal sequence can be directly obtained from the base exponent a (i.e., if $a = 2$ the optimal multiplication sequence is always 2, 4, 8, 16, etc. in this case).

The verifier also computes $k \equiv (a^t + v \cdot M) \bmod \phi(N)$. Once the prover finishes computing Y_1, it is requested to compute $Y_2 \equiv X^k \bmod N$ and report the result back to the verifier. It is known that:

$$X^k \; mod \; N = X^{(a^t + v \cdot M)} \; mod \; N = (Y_1 \cdot m^v) \; mod \; N$$

As we show later, this property will be useful in efficiently verifying the result of our proposed puzzle. The verifier then compares Y_2 to $(Y_1 \cdot m^v) \bmod N$. If this verification passes, the verifier accepts the puzzle solution reported by the prover.

To ensure the security of our scheme, the value of $v \cdot M$ should be different for all the puzzles that are generated using one set of values (e.g., a, t, N, M, for the reasoning why, see Section III-D). This is achieved as follows; the puzzles are generated in sets, where each set $\mathbb{G}$ contains $(2^L - 2)$ distinct puzzles, where L is the size of v in bits. For each set, a new M is chosen (we explain how below) and is used for all the puzzles within a set, i.e., $M^{g_i} = M^{g_j}, \forall i, j \in [1, 2^L - 1]$, $\forall g$, where g is the index of the puzzle set and i, j are puzzle indexes within this set. For each puzzle set with index g, $v^{g_1}, ..., v^{g_{2^L-1}}$ are chosen randomly such that $v^{g_i} \neq v^{g_j}, \forall i, j \in [1, 2^L - 1], i \neq j$; note that $v^{g_i} \in [1, 2^L - 1]$, meaning that $v^{g_1}, ..., v^{g_{2^L}}$ is a

Verifier		Prover
Pre-computation Phase:		
Pick a generator X and a large integer M		
Given p, q, pre-compute $\phi(N) = (p-1)(q-1)$		
Pre-compute $m \equiv X^{M \bmod \phi(N)} \bmod N$		
Start of Evaluation Phase		
Pick a large integer t		
Pick a small *unique* integer* $v \in [1, 2^L - 1]$		
$m_A \leftarrow \{X \| t \| N\}$		
Compute $k \equiv (a^t + v \cdot M) \bmod \phi(N)$		
$m_B \leftarrow \{X \| k \| N\}$		
	$\xrightarrow{m_A \| m_B}$	
		Compute $Y_1 \equiv X^{a^t} \bmod N$
		Compute $Y_2 \equiv X^k \bmod N$
	$\xleftarrow{Y_1 \| Y_2}$	
Verify that $Y_2 = (Y_1 \cdot m^v) \bmod N$		
If the verification passes, the verifier accepts the puzzle solution.		
End of Evaluation Phase		
After a maximum of $\sqrt{2^L}$ different queries:		
$v_{max} \leftarrow 2^L$, where L is the size of v in bits,		
$M \leftarrow v_{max} \cdot M$		
$m \leftarrow m^{v_{max}} \bmod N$		

Fig. 1. Verification-Efficient Puzzle based on Repeated-Squaring. * v is chosen according to the procedure described in Section III-B.

random permutation of the set $[1, 2^L - 1]$. Given this, in each set g, $v^{g_i} M^{g_i} \neq v^{g_j} M^{g_j}$ if $i \neq j$.

To keep the values of $v \cdot M$ distinct between different puzzle sets, M needs to be periodically refreshed for each set such that $v^{g_i} M^{g_i} \neq v^{l_j} M^{l_j}, \forall i, j, g, l, i \neq j, g \neq l$. For this, we propose that for each subsequent puzzle set that it generates, the verifier chooses an M which is strictly larger than the largest $v^{g_i} M^{g_i}$ used so far i.e., $M^{g+1} > 2^L M^g$, where $2^L - 1$ is the largest v that can be used. For all practical purposes, we can set $M^{g+1} = 2^L M^g$. Given that M^g is random to the attacker, so will be M^{g+1}.

The aforementioned process is summarized in Figure 2; the same refreshed value of M will be used by the verifier for a maximum of $\sqrt{2^L}$ different puzzles (refer to Section III-D for further details). In the sequel, this is denoted by the *refresh cycle* of M. Note that this entire process incurs a negligible $O(\log(v))$ (~ 20 modular multiplications) modular multiplications[1] on the verifier.

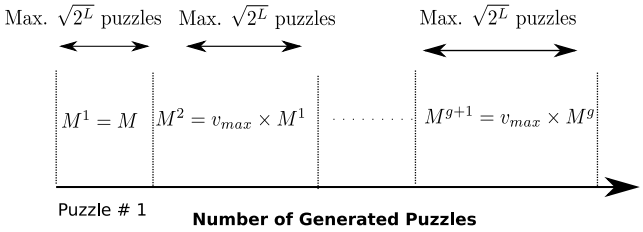

Fig. 2. Periodic Refreshing of the secret M. Here, v_{max} denotes the maximum value of v. That is, $v_{max} = 2^L$, where L is the size of v in bits.

C. Puzzle Construction and Verification Cost

As shown in Figure 1, our proposed puzzle pushes the verification cost incurred in the standard repeated-squaring puzzle [1] to the prover.

We do acknowledge that prime number generation (i.e., computing N) and the pre-computation of m might be computationally expensive for the verifier; however, as mentioned earlier, the verifier re-uses these values in several protocol executions (in typical cases, the verifier pre-computes[2] these values only once). Note that the computational load incurred by prime number generation equally applies to all protocols that make use of modular exponentiation or repeated-squaring (e.g., [1], [2]).

Our solution, however, enables the verifier to perform several rounds of our protocol "on the fly"; indeed, the verifier only needs to perform *one modulus operation* and *one multiplication* (to compute k) to construct the puzzle. Furthermore, the verifier verifies the puzzle solution in a marginal ($O(\log(v))$) number of modular multiplications (comparing Y_1 and Y_2). Given that we use modest values of v (typically 20-bit numbers) in our scheme, the puzzle verification cost can be as low as 20 modular multiplications; this considerably improves the puzzle verification load compared to the standard repeated-squaring puzzle [1]: the

[1]Recall that this equally corresponds to the cost of the verification of our puzzle.

[2]Alternatively, the verifier can be preloaded with the aforementioned values.

	Verifier Cost		Prover Cost	Parallelizable
	Construction	**Verification**		
DH-Based [2]	$O(t)$ mod. mul.	1 comparison	$O(t)$ mod. mul.	Yes
Trapdoor RSA [9]	$O(t)$ mod. mul.	1 comparison	$O(t)$ mod. mul.	Yes
Gaussian Elimination [5]	$O(n)$	$O(n)$ mul.	$O(n^2)$ mul.	Yes
FFT Benchmark [5]	$O(n)$	$O(n)$ mul.	$O(n^2)$ mul.	Yes
Repeated-Squaring [1]	1 modulus 1 mul.	$O(\log(N))$ mod. mul.	$O(t)$ mod. mul.	No
Our Scheme	**1 modulus** **1 mul.**	$O(\log(v))$ **mod. mul.** $^{(*)}$	$O(t) + O(\log(N))$ **mod. mul.**	**No**

TABLE I
COMPARISON OF CONSTRUCTION AND VERIFICATION COSTS OF COMPUTATIONAL PUZZLES. "MOD. MUL." DENOTES MODULAR MULTIPLICATION AND "MUL." REFERS TO MULTIPLICATION. $^{(*)}$ NOTE THAT $v \ll N$; GIVEN A 20-BIT v, THE VERIFICATION COST OF OUR PUZZLE CAN BE AS LOW AS 20 MOD. MUL. WHEN COMPARED TO A VERIFICATION COST OF 1024 MOD. MUL. IN THE ORIGINAL PUZZLE IN [1].

improvement gain is $O(\frac{\log(N)}{\log(v)})$ (e.g., for a 1024-bit N, the average verification cost in the original scheme is approximately $1.5 \cdot \log(N) = 1536$ modular multiplications and the expected improvement gain is almost 51 times).

Note that the verification overhead in our puzzle remains constant even if the security parameters increase over time (e.g., increasing N to 10,000 bits in the future to prevent an attacker from factoring it into p and q). Furthermore, although the verifier periodically refreshes its secret M every $\sqrt{2^L}$ consecutive puzzles ($L \simeq 20$), this only incurs a negligible overhead on the verifier, which corresponds to the verification cost of a single puzzle in our scheme (i.e., $O(\log(v))$ modular multiplications). The refreshing overhead of M also remains constant although the size of M increases after each refresh cycle (in refresh cycle j, $|M^{j+1}| = L + |M^j|$); to refresh M in our puzzle, the verifier computes ${m_j}^{2^L} \ mod \ N$ (Figure 1), which is independent of $|M^j|$.

On the other hand, the prover is required to perform $O(t)$ modular multiplications to compute Y_1 and $O(\log(N))$ modular multiplications to calculate Y_2. Table I compares the puzzle construction and verification costs of our scheme with existing benchmarks and computational client puzzles. Although most of these benchmarks and puzzles can be parallelized, and thus are not recommended to be used in practice, we only include them in our analysis for purposes of comparison.

In what follows, we show that, in spite of its low verification cost, our protocol does not give an advantage to untrusted provers in inflating their puzzle-solving performance claims.

D. Security Analysis

The security of our scheme relies on the fact that it is very hard to obtain $v \cdot M$ from k and therefore to compute $X^{M^v} \bmod N$. This prevents an untrusted malicious prover from tricking the verifier into accepting an incorrect puzzle solution – computed without knowledge of Y_1 and Y_2.

Main Intuition: Given that the factorization of large integers into its prime divisors is very hard, we can argue intuitively that our proposed scheme is secure as follows: M is a secret only known to the verifier, v is picked at random by the verifier and therefore $v \cdot M$ cannot be guessed by the prover. Furthermore, by refreshing the value of the secret M every $\sqrt{2^L}$ consecutive puzzles, $v \cdot M$ always remain distinct and unpredictable across different puzzles. Finally, the construction $v \cdot M$ ensures that the solution space of the ratio $\frac{Y_2}{Y_1}$ is large (comparable to the magnitude of N and therefore cannot be predicted by an untrusted prover) while enabling a small puzzle verification cost of $O(\log(v))$ modular multiplications.

Since $\phi(N)$ is unknown to the provers, $k \equiv (a^t + v \cdot M) \bmod \phi(N)$ further hides the value of $v \cdot M$. Similar approaches are equally used in Zero-Knowledge Proofs [32] (ZKP) to hide secrets [33]. Moreover, although the untrusted prover can compute $\frac{Y_2}{Y_1} = X^{M^v} \bmod N$, it cannot acquire $v \cdot M$ since this discrete logarithm problem is known to be computationally infeasible for large N and M. Given this, we can argue that our protocol does not leak any information that might allow an untrusted prover to factorize $\phi(N)$. One important caveat is that the values of $v \cdot M$ *must* be distinct for each puzzle to ensure the security of our scheme; if $v \cdot M$ is not uniquely chosen, an untrusted prover can obtain, after $O(2^L)$ queries, two different puzzle solutions that were constructed using the same value of v. By performing this attack twice, the attacker can obtain two collisions for the same value of v and subtract the values of k from the number of squarings it performed to get two different multiples of $\phi(N)$. By taking their Greatest Common Divisor (GCD), the attacker might be able to acquire $\phi(N)$. We remedy this attack by choosing distinct v and refreshing the secret M, every $\sqrt{2^L}$ consecutive puzzles.

This aforementioned intuition is developed into a proof below. The proof mainly shows that the security of our proposal is based, to a large extent, on the security of the key generation algorithm of RSA. We show later in this section that the security of our puzzle is equally based on the security of the original time-lock puzzle proposed by Rivest *et al.* in [1].

Theorem *1:* If it is computationally infeasible for an attacker to break the key generation problem in RSA, then the probability that an untrusted prover finds a shortcut to solve our puzzle is satisfactorily negligible.

Proof: Let Π denote the puzzle construction in Figure 1. We show that if there exists a polynomial-time adversary A that can find a shortcut to solve Π (i.e., solve the puzzle without computing Y_1 and/or Y_2), then there exists a polynomial-time algorithm that can find a shortcut to break the key generation problem in RSA; that is, the attacker would then be able to acquire the private key in RSA given the knowledge of the public key e and the modulus N [34].

Let A be a polynomial-time adversary (or malicious prover), and define ϵ as:

$$\epsilon = Pr[A(v \cdot M) = 1]$$

ϵ denotes the success probability of A in obtaining $v \cdot M$ from Π without computing $Y_1 \equiv X^{a^t} \bmod N$ and $Y_2 \equiv X^k \bmod N$. Conforming with the construction of Π, M is a large integer chosen at random and v is a small unique integer chosen at random (typically v is a 20-bit integer). We further assume that A also acquires $\frac{Y_2}{Y_1} = X^{M^v} \bmod N$ (e.g., from prior interactions with the verifier).

Before analyzing the behavior of A, we note that the security of Π relies on the secrecy of $v \cdot M$. Otherwise, an untrusted prover might be able to factor[3] N and acquire $\phi(N)$. This would enable an untrusted prover to send a random value for Y_1 and compute Y_2 accordingly $(Y_1 \cdot X^{(v \cdot M) \ mod \ \phi(N)} \bmod N)$.

Next, we define Γ to be the experiment to break the key generation problem of RSA in polynomial time. Recall that in RSA the private key d is generated from the public key e as follows: $ed - 1 \equiv 0 \ mod \ \phi(N)$, where N is the product of two large primes p and q. In this case, the only shortcut for the attacker to break the RSA key generation problem is to compute $\phi(N)$ given the sole knowledge of the public key e and the modulus N; it is known that this problem is as hard as the factoring problem [34]. Given this, a polynomial-time adversary A can only succeed in breaking the RSA key generation problem with a negligible probability $\bar{\epsilon}$.

In analyzing the behavior of A, we show that the view of A when running sub-routine by Π is distributed identically to the view of A in experiment Γ. We first note the following observations:

1) Due to the hardness of the discrete logarithm problem, A cannot obtain any meaningful information about $v \cdot M$ from $X^{M^v} \bmod N$.
2) The value of $v \cdot M$ is different for each puzzle even if the values of v repeat in each refreshing cycle. By randomly choosing the values of v from a subset $(\sqrt{2^L})$ in $[2, 2^L - 1]$, the probability that A guesses the chosen v for any given puzzle remains satisfactorily negligible even if A guesses by means of elimination all the possible previous values of v (this probability is upper-bounded by $\frac{1}{2^L - \sqrt{2^L} - 2}$)[4]. One direct consequence of this fact is that k and $\frac{Y_2}{Y_1}$, respectively, will not be similar in different puzzles. This shows that an attacker cannot obtain any meaningful information about $v \cdot M$ by observing different values of k and/or $\frac{Y_2}{Y_1}$ (e.g., a birthday paradox involving a collision on $v \cdot M$ is unlikely to happen in this case).

Given this, the only information available to the adversary about v and M in Π is solely contained[5] in k.

In experiment Γ, given a chosen constant C_1, a small unknown unique integer $\bar{v}$ and a large unknown $\bar{M}$, d could be computed as $d = \bar{v} \cdot \bar{M} + C_1$. We define the constant $\bar{C}$ by $\bar{C} = e \cdot C_1 - 1$. Therefore,

Experiment Γ

$$e \cdot d - 1 \equiv 0 \ mod \ \phi(N)$$
$$e \cdot (\bar{v} \cdot \bar{M} + C_1) - 1 \equiv 0 \ mod \ \phi(N)$$
$$e \cdot \bar{v} \cdot \bar{M} + \bar{C} \equiv 0 \ mod \ \phi(N)$$

Since $(\bar{v}, \bar{M})$ and (v, M) have the same distribution (by construction), it is easy to see in this case that if A can obtain $v \cdot M$ from k in Π, it can also obtain $\bar{v} \cdot \bar{M}$ (and therefore d) from Γ; i.e., Π is equivalent to $e \cdot v \cdot M + C \equiv mod \ \phi(N)$, where the constant C is $-e \cdot (k - a^t)$.

This suggests that the probability that A acquires $v \cdot M$ from k in Π is equally $\bar{\epsilon}$. Furthermore, it is easy to see that the only shortcuts for A to solve Π are either to obtain $v \cdot M$ from k (and therefore factorize N) with probability $\bar{\epsilon}$ or to guess the correct ratio $Z = \frac{Y_2}{Y_1}$ with probability $Pr[Z = X^{M^v} \ mod \ N]$. The probability of success ϵ of A in Π is given by:

$$\epsilon = \bar{\epsilon} + Pr[Z = X^{M^v} \ mod \ N] \quad (1)$$

If $\bar{\epsilon}$ is negligible (this is typically the case since breaking the key generation problem in RSA is known to be hard), then $\epsilon \simeq Pr[Z = X^{M^v} \ mod \ N]$. This probability is upper-bounded by the probability that A guesses two "correct" values of v (v_1 and v_2 used in two different puzzles during the same refresh cycle; A then can ensure that the verifier

[3]It is known that if k and $(a^t + v \cdot M)$ are known, then there exists a probabilistic polynomial time algorithm that can compute the factorization of N [34].

[4]The verifier can equally refresh its secret after a larger number of consecutive puzzles e.g., $\frac{v}{2}$. However, in this case, the probability that A might succeed in guessing v equally increases.

[5]A might, however, guess some marginal properties about v and/or M; since 4 is a natural common divisor for both a^t and $\phi(N)$, A might be able to identify from the last two bits of k whether M or v are even/odd or whether they are dividable by 4. However, given large N and M ($\geq$ 1024 bit numbers), this does not give any considerable advantage for an untrusted prover in guessing v or M (if the verifier also varies t in every round, this further increases the randomness of k); this property is equally shared in the original time-lock puzzle of [1] ($a^t \bmod \phi(N)$ is always a multiple of 4) and in Γ ($(e \cdot d - 1)$ is always a multiple of $\phi(N)$).

accepts the puzzle solution by setting $Y_2 = Y_1 \cdot m^{\frac{v_2}{v_1}}$. Therefore, $Pr[Z = X^{M^v} \ mod \ N] \leq \frac{1}{(2^L - \sqrt{2^L} - 2)^2}$. Given this,

$$\epsilon \simeq \frac{1}{(2^L - \sqrt{2^L} - 2)^2}. \quad (2)$$

Note that M remains secret even if A succeeds in guessing one "correct" v and that the attacker has only one chance to guess the value of v (it is uniquely chosen within each refreshing cycle); for $L = 20$ bits, $\epsilon \simeq 9.11 \cdot 10^{-13}$.

This implies that the probability to find a shortcut to solving our proposed puzzle (i.e., without computing Y_1) is satisfactorily negligible, thus concluding our proof. ■

Corollary *1:* The probability that an untrusted prover finds a shortcut to solve the puzzle construction outlined in Figure 1 is satisfactorily negligible if it is infeasible for an attacker to find a shortcut to solve the original time-lock puzzle in [1].

Proof: Let the construction Π and the polynomial-time adversary A be as defined in Theorem 1. Here, we define Γ to be the experiment to break the original time-lock puzzle of [1] in polynomial time. Γ equally models the variant construction of the pseudo-random generator proposed Blum *et al.* in [30]. Recall that in experiment Γ the prover computes $\bar{Y} \equiv X^{a^t} \bmod N$ while the verifier computes $Y \equiv X^{a^t \ mod \ \phi(N)} \bmod N$. If $\bar{Y} = Y$, the verifier accepts the puzzle solution. In this case, the only way for an untrusted prover to break Γ is to compute $\bar{k} \equiv a^t \bmod \phi(N)$ and then $\bar{Y} \equiv X^{\bar{k}} \bmod N$. We denote by $\bar{\epsilon}$ the success probability of a polynomial adversary A in breaking Γ.

Like the proof in Theorem 1, the view of A when running sub-routine Π is distributed identically to the view of A in experiment Γ (refer to Theorem 1 for more details). To better illustrate this, it suffices to recall that $\bar{k}$ can be easily computed as the product of a small secret by a large random secret (i.e., $\bar{k} = v \cdot X + C$, for an integer X given a constant C) and to compare the number of unknown random operands and known operands on both sides of each of the following equalities:

Subroutine Π	**Experiment** Γ
$k \equiv (a^t + v \cdot M) \ mod \ \phi(N)$	$\bar{k} \equiv a^t \ mod \ \phi(N)$
	$v \cdot X + C \equiv a^t \ mod \ \phi(N)$
$v \cdot M \equiv (k - a^t) \ mod \ \phi(N)$	$v \cdot X \equiv (a^t - C) \ mod \ \phi(N)$

Therefore, the probability of acquiring $v \cdot M$ from k in Π is equally the probability of obtaining $v \cdot X$ in Γ (and therefore $\bar{k}$) which is $\bar{\epsilon}$. Similar to Theorem 1, it can be easily shown that the probability for A to find a shortcut to solve our proposed puzzle is given by $\epsilon = \bar{\epsilon} + Pr[Z = X^{M^v} \ mod \ N]$, where $Pr[Z = X^{M^v} \ mod \ N]$ is the probability that A guesses $Z = \frac{Y_2}{Y_1}$. If $\bar{\epsilon}$ is negligible, then $\epsilon \simeq Pr[Z = X^{M^v} \ mod \ N] \simeq \frac{1}{(2^L - \sqrt{2^L} - 2)^2}$.

As mentioned earlier, this probability is satisfactorily negligible for typical values of v ($\epsilon \simeq 10^{-12}$). ■

Claim *1:* An untrusted prover cannot acquire the correct value of $Y_1 \equiv X^{a^t} \ mod \ N$ without performing at least $O(t)$ sequential modular multiplications.

Proof Sketch: Repeated-squaring is an inherently sequential process. The fastest known algorithm for repeated-squaring is the *addition-chain* algorithm [31]. It is expected to perform slightly better than the *doubling algorithm* [35]. However, both algorithms have the same asymptotic running time of $O(\log(a^t)) = O(t)$ modular multiplications. Recall that in our protocol, the optimal multiplication sequence using addition-chains can be efficiently obtained (e.g., 2, 4, 8, 16, etc. if the base exponent $a = 2$).

Although an untrusted prover might try to parallelize repeated-squaring, the parallelization advantage is expected to be very negligible [1], [4]. Since repeated-squaring eventually reduces to a series of sequential modular multiplications, the untrusted prover might try to parallelize the multiplication of large numbers by splitting the multiplicands into smaller "words" and involving other processors in the multiplication of these words. Further details about this process can be found in [36]. However, this attack incurs a significant communication overhead that prevents an untrusted prover from gaining any substantial speedup; given a large number of squaring rounds, the RTT between the cooperating processors needs to be in the order of few nanoseconds to achieve even a modest speedup.

Furthermore, the only known shortcut to compute $Y_1 \equiv X^{a^t} \bmod N$ using less than $O(t)$ modular multiplications is by computing $Y_1 \equiv X^b \bmod N$, where $b \equiv a^t \bmod \phi(N)$. To do so, the prover has to be able to factorize N into its large primes p and q, which is very difficult to achieve even for randomly chosen p and q. Alternatively, the prover might try to guess $\phi(N)$ from k (this probability is given by $O(\frac{1}{N - \sqrt{N}})$, since N is a composite number [37]) or to predict the outcome of $\frac{Y_2}{Y_1}$. Given Theorem 1, the probability that an untrusted prover succeeds in these attacks is satisfactorily negligible.

An untrusted prover can equally try to compute Y_1 through an intermediate known large number that it previously computed. For instance, the prover might use the result of $X^{a^{500,000}} \bmod N$ that it previously computed (e.g., offline or during a subsequent interaction with the verifier) to compute $X^{a^{800,000}} \bmod N$ and thus minimizing the required number of modular exponentiations. Note that this equally applies to the original time-lock puzzle proposed in [1]; it can be simply remedied by changing the

# Squaring (Size of t in bits)	Puzzle Runtime	Y_2 Computation Time	v	Verification	Prob. Cheating
6500000	154067 ms	24.27 ms	10^5	17 mod. mul.	10^{-10}
6500000	172174 ms	27.12 ms	10^7	24 mod. mul.	10^{-14}
6500000	170611 ms	26.877 ms	10^9	30 mod. mul.	10^{-18}
6500000	165034 ms	26 ms	10^{11}	37 mod. mul.	10^{-22}

TABLE II

IMPLEMENTATION RESULTS ON FOUR DIFFERENT INTEL CORE 2.20 GHZ PROCESSORS. HERE, N IS 1024-BIT COMPOSITE INTEGER, "PROB. CHEATING" REFERS TO THE PROBABILITY THAT AN UNTRUSTED PROVER RANDOMLY GUESSES THE RATIO $\frac{Y_2}{Y_1}$. WE CONDUCTED OUR MEASUREMENTS OVER THE LAN (MAX RTT = 100 MS). OUR RESULTS ARE AVERAGED OVER 10 DISTINCT MEASUREMENTS. THE AVERAGE IMPROVEMENT GAIN IN PUZZLE VERIFICATION WHEN COMPARED TO ORIGINAL REPEATED-SQUARING PUZZLE IN [1] IS $\frac{\log(N)}{\log v} \simeq 50$.

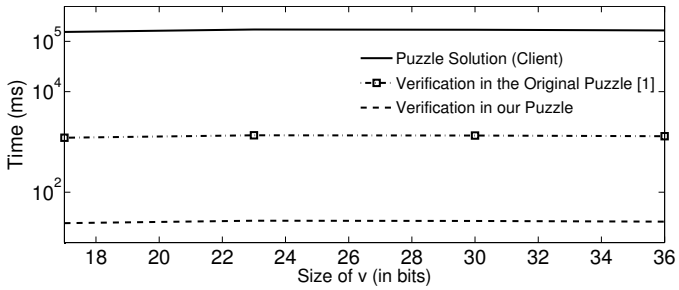

Fig. 3. Puzzle Solution and Verification with respect to the size of v. Our implementation was conducted on an Intel(R) Core(TM)2 Duo CPU T7500 processor running at 2.20 GHz.

base exponent[6] a after each interaction and/or by varying the range of t among different puzzles. On average, the prover still has to perform $O(t)$ modular multiplications. □

Following from Theorem 1 and Claim 1, we conclude that an untrusted prover cannot trick the verifier into accepting a puzzle solution without performing at least $O(t)$ modular multiplications.

E. Evaluation Results

To evaluate the security and performance of our puzzle, we implemented our protocol (Figure 1) in JAVA on four different workstations[7] equipped with Intel(R) Core(TM)2 Duo CPU T7500 processor running at 2.20 GHz. We evaluate the performance of our proposed scheme on various other processors in the following section. In our implementation, we used built-in JAVA functions for prime number generation, repeated-squaring using addition chains, etc.. While a faster implementation of our scheme could be achieved using lower-level programming and/or specialized hardware or software, we aim to demonstrate in this work the feasibility of our proposal using available standard algorithms and programming tools.

Our findings are summarized in Table II. Our results suggest that our scheme establishes a strong tradeoff between the security and the efficiency of puzzle verification. By appropriately choosing the protocol parameters, the verifier can achieve the desired tradeoff that best suits its intended application. For example, by increasing the range of v (e.g., to 10^{12}), the verifier reduces the probability that an untrusted prover randomly guesses the correct v. However, this would equally incur an increase in the computational load on the verifier ($O(\log(v))$). Nevertheless, the choice of any modest value of v (e.g., $v \leq 10^{12}$) does not give any considerable advantage to an untrusted prover in breaking our protocol while incurring a negligible verification overhead on the verifier (Figure 3).

[6]Rivest *et al.* argued that even the choice of a fixed value for the base exponent a does not deteriorate the security of puzzles based on repeated-squaring [1].

[7]We point that our results do not depend on the computational performance of a specific processor (refer to Section IV-A).

IV. APPLICATIONS

This section highlights some applications that would benefit from our scheme.

A. Remote Verification of Computing Performance

To cope with the advances in processing power, the computing community is heavily relying on the use of benchmarks. Benchmarks (e.g., Whestone [38], Linpack [27]) refer to an artificial program/code that is run on a computer system to evaluate its performance (processing speed, access time, etc.). These benchmarks are typically used to evaluate the performance of a single processor unit. While several benchmarks [38], [27], [4], [5] were proposed as a mean to evaluate a processor's computing power, most of these benchmarks are inherently parallelizable (refer to Section II) and therefore not useful to securely evaluating the computing performance of processors (provers can involve other machines at their disposal in the computation of the benchmark solution). Furthermore, current proposals require considerable computational load to verify the benchmark results, which might hinder their prospective large-scale deployment in existing online applications.

Based on our puzzle, we construct a secure benchmark that enables any machine (even with modest computation power) to remotely *upper-bound* the computing performance of single-processors. Our benchmark and the related protocol are shown in Figure 4.

Our benchmark consists of two phases: the *evaluation* phase and the *verification* phase. The verifier constructs the puzzle (as explained in Section III) and measures the time $(t_2 - t_1)$ required by the prover to compute Y_1 in

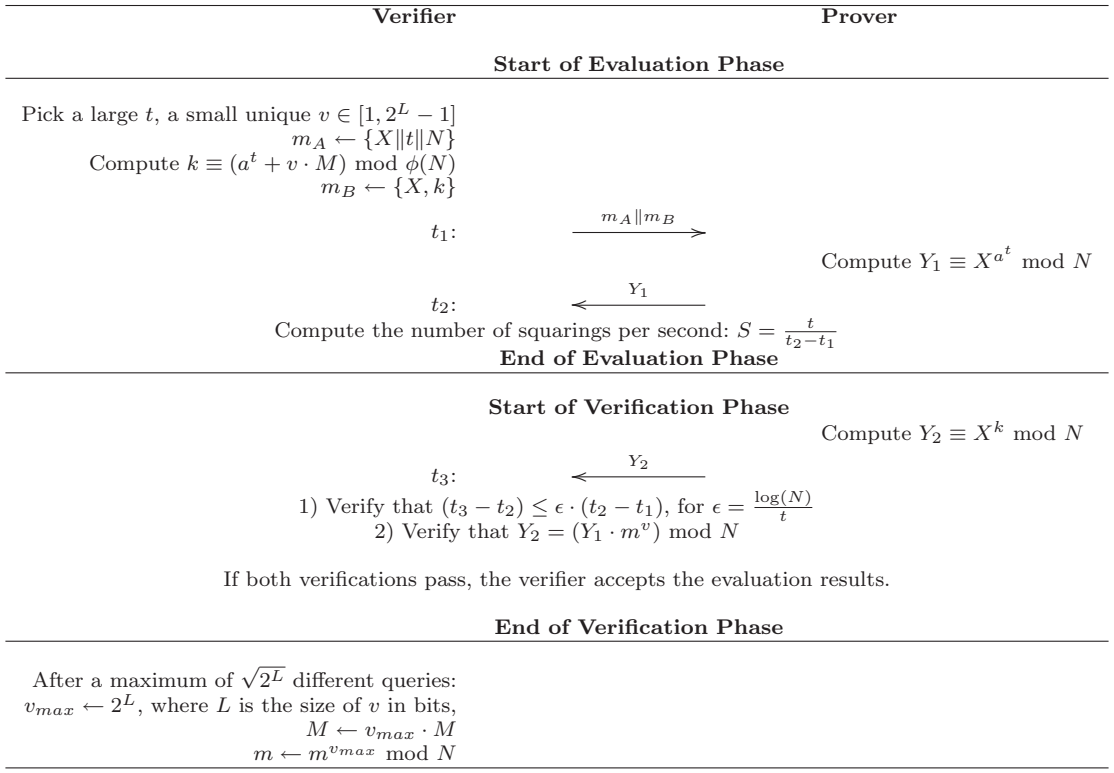

Fig. 4. Efficient Verification of Computing Performance.

order to estimate its puzzle solving performance given by the number of squarings per second: $S = \frac{t}{t_2 - t_1}$.

The verification stage then starts *as soon as* the latter phase is completed. As described later, this is essential to the security of our benchmark. During the evaluation phase, the prover computes Y_2 and reports its result back to the verifier at time t_3. The verifier checks if $Y_2 = (Y_1 \cdot m^v) \bmod N$ and that the time taken by the prover to compute Y_2 is smaller than the maximum threshold $\epsilon = \frac{\log(N)}{t}(t_2 - t_1)$. If both these verifications pass, the verifier accepts S as an authentic estimation of the sequential computing performance of the prover.

Claim *2:* Given the benchmark depicted in Figure 4, the probability that an untrusted prover tricks the verifier into accepting a computing performance claim *inflated* by more than a small ϵ of its genuine value is very negligible.

Proof Sketch: Following from our previous analysis in Theorem 1 and Claim 1, the probability that an untrusted prover can trick the verifier into accepting an incorrect performance claim without performing at least $O(t)$ modular multiplications is satisfactorily negligible.

Therefore, the only viable method for the prover to inflate its performance claim is by sending an incorrect value $\bar{Y_1}$ ahead of time and distributing the computation of the corresponding $\bar{Y_2}$ (such that $\frac{\bar{Y_2}}{\bar{Y_1}} = m^v \ mod \ N$) to other nodes at its disposal. This would enable the prover to decrease the claimed time to perform $O(t)$ modular multiplications by $\Delta = (t_3 - t_2)$ time units (Δ includes the communication delay D between the verifier and the prover).

The verifier can upper-bound $(t_3 - t_2)$ to prevent the attacker from gaining any advantage from the aforementioned attack; the verifier does not accept the prover's performance claim unless $(t_3 - t_2) \leq \epsilon \cdot (t_2 - t_1) + D$. In this case, the maximum performance claim that an untrusted prover can make is:

$$S_{max} = \frac{t}{(1 - \epsilon) \cdot (t_2 - t_1) - D}$$

ϵ can be interpolated from the measured number of squarings per second S; if it takes $(t_2 - t_1)$ time units for the prover to perform t modular multiplications, then the computation of Y_2 (requiring $O(\log(N))$ modular multiplications) can be upper-bounded by choosing $\epsilon = \frac{\log(N)}{t}$. For a 1024-bit modulus N and $t > 100,000$, $\epsilon \simeq 0.01$ squarings per second.

In an experiment that we conducted using 1024-bit N, $t = 6,500,000$, $(t_2 - t_1) = 154067$ ms, a maximum allowable D= 1000 ms (e.g., communication over the Internet), the maximum gain that a prover can claim[8] in our scheme is given by : $\frac{S_{max}}{S}$= 1.0008. □

[8]Note that the verifier cannot ensure that the puzzle is running on the prover's machine. The prover might, for example, rent the fastest machine to perform the evaluation. This, nevertheless, still suggests that the prover *has access* to a machine that can perform S squarings per second.

CPU Description	Idle CPU	S
Intel(R) Pentium(R) 4 CPU 3.40GHz	2.70%	5.9
Intel(R) Pentium(R) D CPU 3.20GHz	6.40%	7.48
AMD Athlon(tm) 64 Processor 3200+	2.60%	12.19
Intel(R) Pentium(R) D CPU 3.00GHz	26.20%	15.24
Intel(R) Pentium(R) 4 CPU 3.20GHz	30.70%	15.81
Intel(R) Pentium(R) D CPU 3.40GHz	14.10%	18.22
Intel(R) Xeon(R) CPU 3060 2.40GHz	46.60%	28.01
Intel(R) Pentium(R) D CPU 3.20GHz	20.00%	29.35
Intel(R) Xeon(R) CPU 3075 2.66GHz	19.70%	29.72
Intel(R) Core(TM)2 Duo CPU E6550 2.33GHz	58.70%	30.39
Intel(R) Core(TM)2 Duo CPU E6550 2.33GHz	60.00%	31.18
Intel(R) Pentium(R) Dual CPU E2180 2.00GHz	66.30%	31.7
Intel(R) Pentium(R) 4 CPU 3.06GHz	92.00%	31.72
Intel(R) Core(TM)2 Duo CPU E6550 2.33GHz	63.80%	36.05
Intel(R) Core(TM)2 Duo CPU T7500 2.20GHz	76.00%	38.11
Intel(R) Xeon(R) CPU X3220 2.40GHz	73.30%	41.67
Intel(R) Xeon(R) CPU E5420 2.50GHz	63.80%	45.59
Intel(R) Xeon(R) CPU E5420 2.50GHz	87.70%	50.97

Fig. 5. Implementation Results on 18 different PlanetLab Nodes. S refers to the number of squarings per ms.

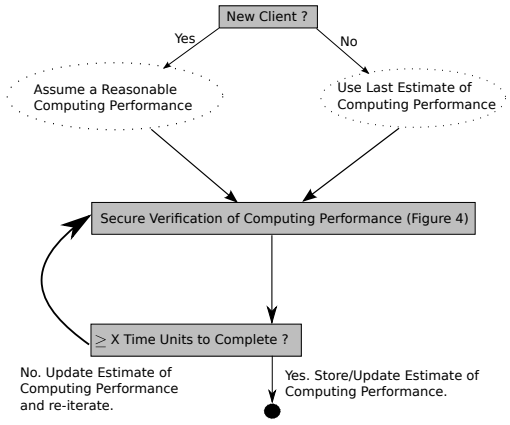

Fig. 6. Efficient Resistance to DoS Attacks through Verification of Computing Performance.

Note that our benchmark does not aim at preventing performance deflation attacks. To the best of our knowledge, it is hard if not impossible to prevent the provers from claiming lower computational performance in the absence of tamper-proof hardware/software.

Nevertheless, our protocol finds clear applicability in a multitude of application domains. Our benchmark can be used in online distributed computing applications (such as [39], [40]) to e.g., enable a participant to verify that it acquired the computing power that it actually asked for. Similarly, our proposal can be used in the secure ranking of supercomputers (e.g., [41]) to prevent possible frauds in performance claims. For instance, Linpack [27] is currently being used in evaluating and ranking the performance of supercomputers [41]. Since Linpack is a benchmark based on Gaussian Elimination, its underlying operation can be easily parallelized [25]. A supercomputer, connected to a hidden processor cluster, can inflate its performance claims by involving these other processors in the construction of the benchmark's solution. The literature contains a significant number of similar "anecdotes" where both individuals and manufacturers have tendencies to exaggerate[9] their computing performance (e.g., [6], [7]). Our protocol is equally well suited for scenarios in which low-end devices (e.g., PDAs or sensor nodes) verify the computational performance of high-end processors. Our protocol can thus be used e.g., to secure performance-based cluster selection in a heterogeneous wireless network.

We evaluated our benchmark on various processors[10] running on 18 different PlanetLab nodes [11] (refer to Section III-E for implementation details). Our findings (Table 5) suggest that our proposed benchmark reflects well the performance of various processors. We acknowledge, nevertheless, that several other factors need to be taken into account when evaluating the overall performance of processors. In Section V, we briefly discuss the limitations and challenges that arise in the estimation of computing performance.

[9]One example cited in [8] was a claim by a scientist that his research was performed on a 65,536-processor computer. Under questioning, the author admitted he had used a system with only 8,192 processors, and then had multiplied his performance figures by a factor of eight.

[10]For better comparison purposes, note that an Intel Xeon processor performs faster than a Pentium D processor, which is in turn faster than a Pentium 4 processor, provided that these processors share comparable properties, such as: CPU architecture, clock frequency, idle CPU %, cache size, etc..

B. Efficient Resilience to DoS Attacks Using Puzzles

Another natural application for our proposed protocol lies in the area of protection against DoS attacks. As mentioned earlier, in existing client-puzzles, it is extremely difficult to precisely specify the time and/or resources required to solve a puzzle, mostly due to the heterogeneity of computing devices available today [10].

Given our puzzle outlined in Section 1, we propose to improve the efficiency of DoS protection schemes based on client puzzles by additionally evaluating/verifying the clients' puzzle solving performance.

Consider the protocol shown in Figure 6; to prevent DoS attacks, an online server requires that its clients spend X time units in solving a puzzle before attending each of their requests. Whenever a new client issues a request, the server uses our performance verification protocol in Figure 4 and sets the protocol parameter t such that the puzzle can be solved in X time units assuming a reasonable computing power. If the client solves the puzzle in $Y < X$ time units, the server re-estimates the client's computing performance (using the protocol in Figure 4) and adjusts the parameter t such that the new computing puzzle can be solved in $(X-Y)$ time units. This process repeats until the client commits X time units to solve the puzzles. On the other hand, if the client solves the puzzle in $Y \geq X$ time units, the process terminates and the server re-adjusts its estimate of the client's computing performance accord-

ingly. The server only needs to evaluate the computing performance of its clients once; in subsequent requests from the same client[11], it re-uses the last estimate of the client's performance (obtained from the last puzzle it solved) to appropriately adjust the protocol parameter.

This approach enables the server to efficiently ensure that all clients – whether running on slow or fast machines – spend enough time and resources before attending to their requests. That is, the server can enforce *fairness* between its clients, e.g., both fast and slow clients can back-off for the same amount of time before their requests are processed by the server.

Given Claim 2, the only viable option for a malicious attacker is to intentionally delay its response to the server in an attempt to prevent the latter from correctly estimating its performance. However, this attack can only penalize the attacker, itself; that is, the attacker will loose more time before its request is being served, thus achieving the intended goal of our scheme[12].

V. Discussion

In the previous sections, we presented and analyzed an efficient puzzle based on repeated-squaring and we demonstrated its usefulness in several security-critical applications. While we acknowledge that our puzzle, based on repeated-squaring, might not be ideal to compare the problem-solving performance of processors, we argue in this section that our proposal presents one of the very few alternatives to *securely* and *efficiently* assess the computing performance of devices.

Nowadays, the literature features increasing efforts to define appropriate metrics to measure performance. Researchers have long argued that "FLOPS" and "MIPS" do not accurately reflect the problem solving performance of a processor [43]; current trends in the literature suggest that the only accurate way to measure the performance of a processor is to test it with respect to those software applications that it is intended to frequently run [44]. This suggests that benchmarks need to be created for each individual application (e.g., Gaussian Elimination benchmark, FFT benchmark, database benchmark, etc.) since several factors need to be considered when assessing the performance of a processor in a given application (e.g., CPU architecture, cache size, existence of a specific hardware, etc.).

When the applications in question can be parallelized (e.g., Gaussian Elimination), little can be done to securely estimate their sequential performance through a puzzle that truthfully mimics the operation of these applications. Such parallelizable puzzles do not even constitute ideal candidates to verify the performance of *parallel computations*; it is generally preferable in these settings to make use of "embarrassingly-parallel" problems (e.g., hash-reversal puzzles [3]) whose parallelization across processors is inherent. Furthermore, memory-bound puzzles cannot be used to assess the computing performance of a device; they can be used however to evaluate the device's memory access speeds.

Moreover, puzzles should be designed such that they contain a shortcut – only known to the verifier – that would enable efficient puzzle verification. This is not the case in most sequential problems. For example, consider the problem of searching for the Great Internet Mersenne Prime [45]. Although this problem is inherently sequential, the only means for the verifier to check whether the puzzle solution is correct is to either run it itself or to sample check intermediate computations (the verifier can also try to parallelize a subset of these computations). Even if the verifier checks a sample of the prover's computations, the puzzle verification overhead would still be considerably large ($O(n)$) to ensure a satisfying detection rate of possible malicious behavior.

Our proposal offers on the other hand several advantages over other benchmarks/puzzles:

- Since our protocol is based on repeated squaring, it is inherently sequential and, therefore, cannot be parallelized. Furthermore, the fastest optimized algorithms implementing repeated-squaring (e.g., addition chains [31]) are known in the literature and available as open-source software in several programming languages. Asymptotic runtime bounds for repeated-squaring are equally established. This gives negligible benefit for a prover in further optimizing the protocol to enhance its puzzle-solving performance.
- Our puzzle offers a shortcut based on number theory (Euler's function) – only known to the verifier – to efficiently verify the puzzle solution using low computational overhead.
- The puzzle parameters can be easily tuned to adjust the difficulty/runtime of the puzzle and the verification load on the verifier. This would facilitate its use in practical real-time applications; our protocol enables the verifier to ensure that the provers spend enough time and resources, in spite of the heterogeneity of their machines.
- Although our puzzle might not faithfully emulate the performance of a processor in all existing applications (e.g., solving linear equations, sorting lists), it can be used to evaluate the computing performance in several applications that rely on modular exponentiation (e.g., cracking asymmetric decryptions, SETI@home [39], etc.)
- Our scheme can also be used to securely outsource exponential computations (e.g., public key encryptions) to untrusted servers.

[11]It is out of the scope of this paper to discuss algorithms and techniques that enable a remote server to identify its clients. Further details on this topic can be found in [42]. For simplicity, we assume that the server identifies remote nodes using their IP addresses. Owing to its low verification overhead, the server can efficiently re-evaluate a node's computing power in case it changes its public credentials (e.g., IP renewal).

[12]It can be shown that our scheme can equally be used to counter connection depletion attacks.

VI. Conclusion

In this work, we proposed a new verification-efficient client puzzle based on repeated squaring. Our puzzle extends the time-lock puzzle proposed by Rivest *et al.* and enables a considerably more efficient verification of the puzzle solution that is reported by provers. More specifically, our scheme transfers the puzzle verification burden to the prover that executes the puzzle; we achieve this by embedding a secret – only known to the verifier – within the Euler trapdoor function that is used in repeated-squaring puzzles. Given this, the improvement gain in the verification overhead of our puzzle when compared to the original repeated-squaring puzzle is almost 50 times for a 1024-bit modulus. We further showed how our puzzle can be integrated in a number of protocols, including those used for protection against DoS attacks and for the remote verification of the computing performance of devices. We provided a security proof for our scheme and we validated the performance of our puzzle through experiments on a large number of PlanetLab nodes.

References

[1] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release crypto," in *MIT Technical Report*, 1996.

[2] B. Waters, A. Juels, J. A. Halderman, and E. W. Felten, "New client puzzle outsourcing techniques for dos resistance," in *Proceedings of the ACM conference on Computer and Communications Security*, 2004.

[3] S. Tritilanunt, C. Boyd, J. M. G. Nieto, and E. Foo, "Toward non-parallelizable client puzzles," in *Proceedings of CANS*, 2007.

[4] R. Sedgewick and A. C.-C. Yao, "Towards uncheatable benchmarks," in *Proceedings of The Structure in Complexity Theory Conference*, 1993.

[5] J. Cai, A. Nerurkar, and M. Wu, "The design of uncheatable benchmarks using complexity theory," available from ftp://ftp.cs.buffalo.edu/pub/tech-reports/./97-10.ps.Z.

[6] Conroe Performance Claim being Busted, http://sharikou.blogspot.com/2006/04/conroe-performance-claim-being-busted.html.

[7] Computer Software Manufacturer agrees to settle Charges, http://www.ftc.gov/opa/1996/07/softram.shtm.

[8] Heidelberg Talk Tells How toFool the Masses, http://www.lbl.gov/cs/Archive/news062904.html.

[9] Y. Gao, "Efficient trapdoor-based client puzzle system against dos attacks," 2005.

[10] P. Tsang and S. Smith, "Combating spam and denial-of-service attacks with trusted puzzle solvers," in *Proceedings of the Information Security Practice and Experience Conference*, 2008.

[11] PlanetLab, An open platform for developing, deploying, and accessing planetary-scale services, http://www.planet-lab.org/.

[12] X. Wang and M. K. Reiter, "A multi-layer framework for puzzle-based denial-of-service defense," in *International Journal of Information Security*, 2007.

[13] X. Wang and M. Reiter, "Defending against denial-of-service attacks with puzzle auctions," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2003.

[14] A. Juels and J. Brainard, "Client puzzles: A cryptographic countermeasure against connection depletion attacks," in *Proceedings of NDSS*, 1999.

[15] A. Back, "Hash cash - a denial of service countermeasure," in *Technical report*, 2002. [Online]. Available: http://www.hashcash.org/

[16] S. Doshi, F. Monrose, and A. Rubin, "Efficient memory bound puzzles using pattern databases," in *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, 2006.

[17] D. Dean and A. Stubblefield, "Using client puzzles to protect tls," in *Proceedings of the USENIX Security Symposium*, 2001.

[18] A. Martin, M. Burrows, M. Manasse, and T. Wobber, "Moderately hard, memory-bound functions," in *ACM Transcations on Internet Technologies*, 2005.

[19] M. J. Atallah, K. N. Pantazopoulos, J. R. Rice, and E. H. Spafford, "Secure outsourcing of scientific computations," in *Advances in Computers*, 2001.

[20] W. Du, J. Jia, M. Mangal, and M. Murugesan, "Uncheatable grid computing," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[21] M. Jakobsson and S. Wetzel, "Secure server-aided signature generation," in *Proceedings of the 4th International Workshop on Public Key Cryptography (PKC), LNCS, Springer*, 2001, pp. 383–401.

[22] M. van Dijk, D. Clarke, B. Gassend, G. E. Suh, and S. Devadas, "Speeding up exponentiation using an untrusted computational resource," in *Designs, Codes and Cryptography*, vol. 39, 2006, pp. 253–273.

[23] S. Hohenberger and A. Lysyanskaya, "How to securely outsource cryptographic computations," in *Theory of Cryptography Conference, LNCS, Springer*, vol. 3378, 2005, pp. 264–282.

[24] L. Keqin, "Scalable parallel matrix multiplication on distributed memory-parallel computers," in *Proceedings of IPDPS*, 2000.

[25] S. McGinn and R. Shaw, "Parallel gaussian elimination using openmp and mpi," in *Proceedings of the International Symposium on High Performance Computing Systems and Applications*, 2002.

[26] Z. Cui-xiang, H. Guo-qiang, and H. Ming-he, "Some new parallel fast fourier transform algorithms," in *Proceedings of Parallel and Distributed Computing, Applications and Technologies*, 2005.

[27] Linpack, http://www.netlib.org/linpack/.

[28] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of ACM CCS*, 2003.

[29] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proceedings of the USENIX Security Symposium*, 2006.

[30] L. Blum, M. Blum, and M. Shub, "A simple unpredictable pseudo-random number generator," in *SIAM J. Computing*, 1986.

[31] N. Koblitz, "A course in number theory," 1987.

[32] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," in *Proceedings of the 17th Symposium on the Theory of Computation*, 1985.

[33] C. Schnorr, "Efficient identification and signatures for smart cards," in *Advances in Cryptology*, 1989.

[34] J. Katz and Y. Lindell, "Introduction to modern cryptography," in *Chapman and Hall/CRC Press*, August 2007, pp. 273–274.

[35] D. Knuth, "The art of computer programming," vol. 2, 1969.

[36] C. K. Koc, T. Acar, and B. Kaliski, "Analyzing and comparing montgomery multiplication algorithms," 1996.

[37] W. Sierpinski, "Elementary theory of numbers," 1964.

[38] H. Curnow and B. Wichman, "A synthetic benchmark," in *Computer Journal*, 1976.

[39] SETI@home, http://setiathome.ssl.berkeley.edu/.

[40] Distributed.Net, http://distributed.net/.

[41] TOP500 Supercomputing Sites, http://www.top500.org/.

[42] T. Kohno, A. Broido, and K. Claffy, "Remote physical device fingerprinting," in *IEEE Transactions on Dependable and Secure Computing*, 2005.

[43] J. R. Rice, "Measuring the performance of parallel computations," in *Proceedigns of the International Workshop on Parallel Processing*, 1996.

[44] Benchmark Limitations, http://www.intel.com/performance/resources/benchmark_limitations.htm.

[45] The Great Internet Mersenne Prime Search, http://www.mersenne.org/prime.htm.