ON PATROL

DATABASES
Spring 2001 - 7.3

Resource Allocation, Proofs of Work, and Consequences

by *David Molnar*

In January 2000, the employees of Yahoo!, eBay, and other web sites received a crash course in the **resource allocation problem**. Thousands of ~~computers that~~ had been broken into by crackers connected to these targeted web servers at once. Because opening a new connection was free, easy, and automated, nothing stood between these computers and complete exhaustion of server resources.

The resulting server failures led two researchers from RSA Data Security to propose a defense [6]. Before failing completely, but after an attack has begun, the server would force clients to perform a **proof of work**, a protocol which guarantees the server that a client has spent a certain amount of its own computing power on a particular problem. By making it "cost" something for a client to open a connection, attacks become too "expensive" for an adversary to carry out. When a computer tries to exhaust server resources, it is unable to compute "proofs of work" faster than the server can make room for new connections.

Proofs of work are an example of how cryptography can be applied to a concrete security application. The cryptography is used to specify a "proof of work" and to find good examples of proofs of work to reduce the effectiveness of denial of service attacks.

This method of using proofs of work to create a "cost" for operations reaches beyond a single application. A world with widespread proofs of work is a world with intriguing possibilities. Among these possibilities is an alliance between proofs of work and services that "sell spare cycles." Another more troubling prospect is proofs of work that cause the client to unwittingly participate in some larger computation, such as the brute-forcing of a cryptographic key. Each of these possibilities leads to a series of security related questions.

Resource Allocation and Denial of Service

Resource allocation is an old problem that pops up in many different guises and in many different settings. Often, there are one or more limited resources that must be allocated to one or more parties.

The Tragedy of the Commons

All of economics can be considered a search for solutions to the resource allocation problem. In fact, there is a lesson from economics, known as the **tragedy of the commons** [4], with direct application to resource allocation on the Internet.

According to the story, an area of grass, called the "commons," was owned by no one and free for all to use. Every day, farmers from the surrounding villages would come to graze their livestock on the commons. Because it cost nothing for the farmers to graze livestock, each of them would graze as much as they wished. As the days rolled past, each farmer noticed that the grass was becoming thinner and thinner, showing evidence of overgrazing. Sometimes a farmer would consider grazing fewer of his livestock on the commons, then cast a wary eye at the neighboring herd. How could he stop grazing his livestock when he had no assurance his neighbor would do the same? This continued until one day no more grass was left and everyone starved to death.

What happened here? Even though the farmers paid nothing to use the commons, each farmer used up some of a precious resource -- the grass on the commons. However, the "cost" of using up this resource was shared between all of the farmers. Rational, self-interested farmers recognized the cost, but continued grazing their livestock because there was no guarantee that the other farmers would follow suit.

The situation is worsened by malicious farmers. For example, if one of the farmers started destroying plots of grass often used by his neighbors, nothing could be done to stop him. The grass is not owned by anyone, and the farmer does not incur a cost for his malicious usage. Similarly, resources online are shared by all and can be wasted by malicious users.

Proofs of Work

One method for managing resources online involves creating a "cost" to the clients for using the resource. By introducing a cost for resources, profligate or malicious use is deterred and the tragedy of the commons is averted.

A straightforward implementation would simply place a monetary cost for each connection or other resource allocated. "Micropayment" digital cash schemes favor this approach. Unfortunately, the Internet currently lacks a stable and widespread payment scheme, other than credit cards; unfortunately, credit cards require significant infrastructure. In addition, consumers tend to dislike pay-per-use access [3].

An alternative approach to adding a cost, first pointed out by Dwork and Naor, asks a client to expend a specified amount of computation time to use a service [2]. The client produces a **proof of work**, which is a short certificate of validating that it spent a quanta of its own time on a hard computational problem. Without a proof of work, the client cannot receive service. The assumption is that since a client is limited in computation power, it cannot create enough proofs of work to flood the server.

A Sample Proof of Work: Hashcash

One of the best-known proofs of work is Back's **hashcash**, which was originally developed for spam prevention [1]. Spam is a problem because a spammer can send many e-mails at once with little or no cost to himself. Back's idea was to make a "cost" or "postage" for e-mail that would be denominated in "hashcash." Hashcash is created by spending computer time on a hard computational problem. For example, one might use hash-cash to affix "postage" to every piece of e-mail. The amount would be sizeable enough that the "cost" would not slow down a user who sends a few e-mails a day, but would be prohibitive for a spammer sending thousands or millions of e-mails. The hard computational problem that hashcash is based on involves finding partial collisions in a collision-resistant one-way hash function.

Some explanation is in order here. A **hash function** is any function that takes an arbitrary length string and returns strings of a fixed length. In symbols, a hash function h is denoted by `h : {0,1}^* --> {0,1}^k` for some fixed constant `k` which determines the output length. In practice, examples of hash functions include RSA Data Security's MD5, which has an output of 128 bits, and the NSA-designed Secure Hash Algorithm (**SHA-1**) with 160 bits. A **collision** in the hash function exists when there are two elements `x` and `x'` such that `h(x) = h(x')`. If the hash function is **collision-resistant**, then it is hard to find any `x` and `x'` that "collide", i.e. `h(x) = h(x')`. A **partial collision** is a pair `y` and `y'` such that `h(y)` agrees with `h(y')` on a certain number of consecutive bits.

In words, if `H()` is a collision-resistant hash function, then given `H(52) = 10001`, it is hard to find anything else which can be plugged into `H()` to give `10001` as an output. Furthermore, it is hard to find two inputs to `H()` which have the same output, even if that output is allowed to be anything at all. A partial collision might be something like `H(52) = 10001` and `H(1923) = 11101`, which agree on two consecutive bits total out of 5.

How hard is "hard?" Ideally, the hash function should look like a **random function**, which is a function whose outputs are all chosen at random. Probability theory tells us that finding a collision for a specific, targeted `x` and `h(x)` should take on average `2^k` evaluations of the function. When a target collision is not at issue, but instead just any collision for `x` and `x'`, only `2^(k/2)` evaluations are required. For a partial collision with `n` bits in common, the corresponding work to find any two values `y` and `y'` is `2^(n/2)` evaluations of the function.

Now a server can specify a particular hash function, such as the U.S. Government's SHA-1, a particular value for n, and a certain number of partial hash collisions it wants to see. This is the cost of using a resource. For example, a mail server might say that it wants to see 4 partial collisions with 64 bits in common each, which will take 4 * 2^(64/2) = 2^(32) operations. By fine tuning these parameters and estimating how much computational power a client has, the server can establish a fair "price" for sending mail.

For example, suppose that a mail server decides it wants to see a 64-bit partial collision attached to each e-mail before it will accept the message for delivery. This means that in order to send one message, a client will have to perform on average 2^32 operations of the SHA-1 hash function in order to find these partial collisions. For the crypto library BeeCrypt, a P3 450MHz running Windows 98 can hash 19.5 MB/sec [11]. Since SHA-1 works on 512-bit blocks [9], one MB/sec translates into 2 * 2^20 operations per second.

Therefore, performing 2^32 operations will take about 105 seconds of time to find a partial collision. By contrast, a spammer with the same computer who wishes to send 100,000 messages has to wait for 105 * 100,000 seconds, which is about 121 days.

Proofs of Work and Distributed Denial of Service

Would a proof of work protocol have stopped the distributed denial of service attack on Yahoo? Sadly, the answer is likely no. Even though a server has to do relatively little work to verify each proof of work, it still must spend some time per client to verify each proof. Once sufficiently many clients connect, verifying their proofs of work can exhaust the host's resources. An adversary mounting a distributed denial of service attack simply recruits more machines, especially machines with large amounts of bandwidth. Still, proofs of work can give a machine some "breathing room" while its owners search for the adversary.

Formalizing Proofs of Work

What's next after hashcash? What other kinds of proofs of work are there? A formal definition can be produced, which is an explicit statement of everything that a proof of work "should have." In fact, Jakobsson and Juels have done just that [5].

Reusing Stale Computation

The hashcash proof of work has an annoying property. The computation used to create hashcash is "wasted" in the sense that it does no useful work beyond making more hashcash. It would be nice if this computation could do something useful while wasting the client's time.

Not only did Jakobsson and Juels produce a formal definition of proofs of work, but they turned their attention to this problem of "reusing computation" as well. They developed an example protocol that uses a client's proof of work to help a server participate in a digital cash scheme, developed by Rivest and Shamir, called MicroMint [8]. In addition, they defined the notion of "bread pudding protocols." These protocols reuse stale computation that would otherwise have been wasted [5].

Possible Consequences of Proofs of Work

This section and the following are speculative in nature, outlining two possible future uses for proofs of work.

Proofs of Work and Selling Spare Cycles

In a previous column, titled "The Seti@Home Project," I noted the existence of startups like centrata.com, that sell the spare cycles of many clients [7]. Proofs of work offer an easy way to recruit clients. One can easily envision a proof of work that asks a client to perform a distributed computation work unit. Imagine if Yahoo! required connecting clients to perform a work unit before every connection.

Proofs of Work and "Distributed Kleptography"

Adam Young and Moti Yung defined "kleptography" as the "art of using cryptography to steal information". As an example, they showed how a cryptographic system could "leak" bits of its secret key in a manner that would be undetectable by the user of the system. Only the adversary could see the "leaked" key bits and recover the key. In this setting, what is being stolen is clearly of value, because possession of a secret key allows the reading of all messages encrypted with that key [12].

In a world where many clients connect to many servers, spare computation cycles become available, salable, and therefore valuable. Consequently, "distributed kleptography" in this setting would involve stealing spare computation.

Does this problem already exist? After all, most versions of Java, while placing strict limits on an applet's disk and network usage, do not explicitly limit an applet's CPU time. Today, however, if an applet uses too much time, it's likely to be killed by a user because applets are not supposed to waste time. A world with ubiquitous proofs of work is a world in which clients are accustomed to wasting time. Combined with a world in which proofs of work are used by services which sell spare cycles, this would make servers a tempting target for viruses that change proof of work requests. In addition, one may postulate the existence of protocols with "subliminal proofs of work", i.e. proofs of work that appear to be computing one hard problem, but instead compute a different problem. For instance, can part of a brute-force search on a cryptographic key be "hidden" in a SETI@Home work unit?

One possible case of "distributed kleptography" was described by Twyman. An adversary uses a virus or a Java applet to steal time on many computers and apply it towards the cracking of a cryptographic key [10].

Conclusion

Proofs of work are an intriguing cryptographic idea with concrete security applications. First proposed to deal with spamming and denial of service attacks, proofs of work are finding applications in surprising places. The formalization of proofs of work, combined with their importance to resource allocation problems, is making them a technology to stay on top of.

Acknowledgements

The author is indebted to Roger Dingledine and Michael Freedman for many discussions on this topic. The article would not be possible without the anonymous reviewers and Crossroads editors.

References

1

Back, A. *Hashcash.* http://www.cypherspace.org/~adam/hashcash/

2

Dwork, C. and Naor, M. Pricing Via Processing. In *Proceedings of CRYPTO '92*. http://www.wisdom.weizmann.ac.il/Dienst/UI/2.0/Describe/ncstr l.weizmann_il/CS95-20

3

Fishburne, P.C. and Odlyzko, A. Competitive pricing of information goods: Subscription pricing versus pay-per-use. In *Economic Theory* 13 (1999), pp. 447-470.

4

Hardin, G. The Tragedy of the Commons. In *Science*, 162 (1968), 1243-1248. http://dieoff.com/page95.htm

5

Jakobsson, M. and Juels, A. Proofs of Work and Bread Pudding Protocols. In B. Preneel, ed. *Communications and Multimedia Security*. Kluwer Academic Publishers, pp. 258-272.

6

Juels, A. and Brainard, J. Client Puzzles: A Cryptographic Countermeasure Against Depletion Connection Depletion Attacks. In *Proceedings of the Network and Distributed Security Symposium - NDSS '99*. http://www.isoc.org/ndss99/proceedings/papers/juels.pdf

7

Molnar, D. The SETI@Home Problem. *ACM Crossroads*. September 2000. http://www.acm.org/crossroads/columns/onpatrol/september2000.html

8

Rivest, R. and Shamir, A. PayWord and MicroMint: Two Simple Micropayment Schemes. Presented at RSA '96 conference. http://citeseer.nj.nec.com/rivest-payword.html

9

Schneier, B. *Applied Cryptography*. John Wiley and Sons, 1996.

10

Twyman, A. *The Threats of Distributed Cracking*. http://www-swiss.ai.mit.edu/6805/student-papers/fall97-papers/twyman-cracking.html

11

Virtual Unlimited. *BeeCrypt Benchmarks*. at http://www.virtualunlimited.com/products/beecrypt/benchmarks.html

12

Young, A. and Yung, M. Kleptography: Using Cryptography against Cryptography. In *Proceedings of CRYPTO '97*. http://home.bip.net/laszlob/cryptoag/kleptography.htm

---

Biography

David Molnar (dmolnar@hcs.harvard.edu) began using PGP in 1993. He became interested (obsessed?) with figuring out "why it worked" and has been studying cryptography ever since. Now an undergraduate at Harvard University, he keeps up with security issues by attending courses, reading newsgroups, mailing lists, and conference papers, and attending DEF CON in his home city of Las Vegas. David is an ACM Student Member and a member of the International Association for Cryptologic Research.
