REMOTE ALGORITHMIC COMPLEXITY ATTACKS AGAINST RANDOMIZED HASH TABLES

Noa Bar-Yosef *
School of Computer Science
Tel Aviv University, Ramat Aviv 69978, Israel
noabary@post.tau.ac.il

Avishai Wool
School of Electrical Engineering
Tel Aviv University, Ramat Aviv 69978, Israel
yash@acm.org

Keywords: Algorithmic complexity attack, denial of service, packet filter.

Abstract: Many network devices, such as routers, firewalls, and intrusion detection systems, usually maintain per-connection state in a hash table. However, hash tables are susceptible to algorithmic complexity attacks, in which the attacker degenerates the hash into a simple linked list. A common counter-measure is to randomize the hash table by adding a secret value, known only to the device, as a parameter to the hash function. Our goal is to demonstrate how the attacker can defeat this protection: we demonstrate how to discover this secret value, and to do so remotely, using network traffic. We show that if the secret value is small enough, such an attack is possible. Our attack does not rely on any weakness of a particular hash function and can work against any hash — although a poorly chosen hash function, that produces many collisions, can make the attack more efficient. We present a mathematical modeling of the attack, simulate the attack on different network topologies and finally describe a real-life attack against a weakened version of the Linux Netfilter.

1 INTRODUCTION

1.1 Background

Many network devices, such as routers, firewalls, and intrusion detection systems, need to maintain per-connection state. One commonly used data structure of choice is a hash table. This choice is mainly based on the fact that in the average case retrieving elements from a hash table takes an expected $O(1)$ operations, independent of the number of connection states. However, in the worst case, a hash table can also degenerate into a linked list, and operate in $O(n)$ steps. Because of this, (Crosby and Wallach, 2003) showed that an attacker can remotely mount an “algorithmic complexity attack” against the hash table: if the attacker knows the victim’s hash function, she can force this worst case behavior by producing a long sequence of items that are placed in the same hash bucket, thereby exhausting the device’s CPU time.

*Supported by the Deutsch Institute

A common counter-measure is to randomize the hash table: this is the approach taken by Netfilter (Filter,). To do so, the hash function calculation includes a secret random value known only to the device. Randomizing the details of the hash calculation is supposed to disable the attacker’s ability to manufacture items that, predictably, fall into the same hash bucket. In this work, we show that randomized hash tables are not necessarily the correct measure to apply, and that under certain circumstances, given enough time and space, an adversary can still force the device’s hash table to perform in its worst case behavior and mount an algorithmic complexity attack.

1.2 Related Work

Our research builds upon the work of (Crosby and Wallach, 2003). In their paper they introduce a family of low-bandwidth denial of service attacks called algorithmic complexity attacks, which exploit algorithms’ worst-case behaviors. The focus of their work is on deterministic hash tables. Using their method,

the adversary can create specific inputs that all fall into the same hash bucket, causing the hash table to degenerate into a simple linked list. They successfully carried out their attacks on different applications, such as the IDS, Bro (Paxson, 1999) and several Perl versions. They showed how within 6 minutes they were able to cause the server to consume all of its CPU as well as to drop most of its received packets.

A low-bandwidth attack differs from other common TCP attacks that exhaust the server's resources, such as memory or bandwidth, resulting in a denial of service (Needham, 1993). A typical (high bandwidth) attack that exhausts the server's backlog queue is the well-known syn-flooding attack (SYN flood, 1996). The attacker basically floods the victim with more traffic than the victim can process.

The difficulty with a low-bandwidth attack is that it is much harder to detect than a flooding attack. The aim in low-bandwidth attacks is not to explode the server's resources in an aggressive manner, but rather to exploit vulnerabilities in the server slowly, culminating in a denial of service. For example, (Kuzmanovic and Knightly, 2003) discuss a low-rate denial of service attack that exploits the retransmission timeout mechanism in TCP. By sending small bursts of packets at just the right frequency, the attacker can cause all TCP flows sharing a bottleneck link to simultaneously stop indefinitely. And because the attacker only needs to burst periodically, the attack traffic will be difficult to distinguish from normal traffic.

The notion of a low-bandwidth attack exploiting an algorithm's worst case can be traced back to an attack using nested HTML tables. Some browsers' algorithms perform super-linear work to determine the layout of the table. Thus, a maliciously crafted web page can cause the browser to freeze (Garfinkel, 1996). (Dean and Stubblefield, 2001) propose a solution to an attack against an SSL server that may lead to the paralysis of e-commerce websites. In their scenario, the attacker requests the server to engage in expensive RSA decryptions without first having done any work. An algorithmic complexity attack can also be performed against the quicksort algorithm as shown by (McIlroy, 1999). Quicksort is a common choice of sorting algorithm because of its expected average-case running time of $O(nlogn)$. However, McIlroy provides a way to force quicksort into achieving its worst case running time of $O(n^2)$. Another example includes the attack presented in (Gal et al., 2004). In their work, the attacker takes advantage of the fact that the Java bytecode verification scales quadratically with the size of the program and so keeps the verifier busy in order to constitute a denial of service. The authors develop this notion in order to construct complexity attacks against mobile-code systems. They show the difficulty of conventional defenses thwarting the attack since the attack not only is located ahead of the point at which run-time resource control sets in but it also attacks the mechanism that ensures safety in regards to the Java Bytecode verifier. It is worthwhile to mention a more recent paper by the aforementioned authors (Gal et al., 2005) where they warn that algorithmic complexity attacks are going to be prevalent on all systems, from mobile code systems, to software applications, and hardware. These authors then advocate a new security paradigm based on complexity-hardened systems.

Our attack strategy involves guessing the secret random value of the hash function parameter. The technique resembles those implemented in timing attacks where the attacker determines a victim's secret by analyzing the victim's processing time remotely over the Internet. For example, (Boneh and Brumley, 2003) devise a timing attack against OpenSSL where the client is able to extract the private key stored on the server by measuring the time the server takes to respond to decryption queries. More recently (Kohno et al., 2005) showed how to fingerprint a device remotely by finding microscopic deviations according to each computer's unique clock skew. A practical timing attack is mentioned in RFC 4418, Message Authentication Code using Universal Hashing (UMAC) (RFC4418,), which warns of a possible timing attack in the UMAC algorithm since the behavior of the algorithm differs according to the length of the inputted string. In (Shacham et al., 2004), the authors show a practical timing attack in which they overcame anti-buffer overflow memory randomization protection techniques. Many operating systems now randomize their initial address space as a way to avoid buffer overflow attacks. However, the authors show a feasible way to find the random value, and thus the address space is calculated in a straightforward manner leaving the system once again vulnerable to buffer overflow attacks. They further investigate various strengthing address-space randomization techniques.

1.3 Contributions

Our starting point is the observation that if the attacker can discover the secret value that is used inside the server's hash function calculation, then she can mount the algorithmic complexity attack of (Crosby and Wallach, 2003). Therefore, our goal is to demonstrate how the attacker can discover the secret value, and to do so remotely, using network traffic. We show

that if the secret value is small enough, such an attack is possible. Our attack does not rely on any weakness of a particular hash function and can work against any hash, including cryptographic hashes[2] — although a poorly chosen hash function, that produces many collisions, can make the attack more efficient.

The attack scenario we envision consists of two stages: (i) An offline calculation and information gathering stage, followed by (ii), a full-blown algorithmic complexity attack. In this paper we focus on the first stage.

Our attack is an exhaustive search performed against all possible choices of the secret value. For each candidate secret value, X_i, we produce a set of packets that would hash to the same bucket, send them to the server, and measure the round-trip time (RTT). If the server's secret value is X_i, then there will be a slowdown in the RTT. After trying all the possible values X_i the one causing the longest RTT is likely to be the correct secret X_i. The challenges we face are: (i) Being able to produce enough attack packets so that RTT slowdown will be significantly longer than normal network RTTs, and (ii), doing so in a way that lets the attacker receive the server's responses, so she can measure the RTT — i.e., without spoofing the source IP address.

We demonstrate, via mathematical analysis, that the attack is plausible. We then conducted a simulation study, followed by an actual implementation of the attack against Netfilter. Both simulations and implementation show that the attack is very realistic for secret values of 13-14 bits using current hardware.

Organization: In Section 2 we describe the algorithmic complexity attack of (Crosby and Wallach, 2003) and the Linux Netfilter. In Section 3 we describe our attack and provide some mathematical modeling about its properties. In Section 4 we describe an attack implementation against a weakened version of the Linux Netfilter stateful firewall. We conclude in Section 5.

[2]The reason is that the number of buckets in the hash table is, intentionally, rather small—the default value for netfilter is 8192 buckets. Even a strong cryprographic hash will produce many bucket collisions once its output is reduced modulo 8192.

2 PRELIMINARIES

2.1 Algorithmic Complexity Attacks Against Hash Tables

In a hash data structure, an item is hashed through a hash function which produces a hash output. The output is then stored in the hash bucket, corresponding to the output modulo the number of buckets in the hash table. Items that hash into the same bucket form a linked list in that hash bucket. In order to retrieve a stored item, the server first computes the hash function to find the correct bucket, and then traverses through the list in the corresponding hash bucket to locate the item. A properly implemented hash function will distribute its inputs evenly throughout the array, creating very short lists in the buckets, so that retrieving a stored item will perform in an $O(1)$ average lookup time. However, if an adversary knows the details of the hash function, can control its input, and if the number of buckets is small enough, then she can produce inputs that all collide into the same hash bucket. In the worst case scenario one will have to traverse a list of all the items stored, resulting in the same lookup time complexity as that of a regular linked list, $O(n)$, assuming n elements were hashed. What (Crosby and Wallach, 2003) did was to demonstrate that an attacker can force such worst-case behavior, over the Internet, against a variety of network devices. If the attacker can cause all the hash lookups to run in $O(n)$ steps — she can waste enough CPU time on the server to create a denial of service condition.

Note that a malicious attacker can induce such worst-case behavior against *any* hash function, including cryptographic hash functions. Our method does not rely on the strength of the algorithm itself, but rather on the search space of the secret input key. Since the input space is much larger than the hash table size, many hash collisions are bound to occur. If the hash function is weak, then the attacker can easily find many inputs with the same hash. But even for an ideal hash function the attacker can run an offline computation and find a large set of items that fall into the same bucket.

2.2 Linux Netfilter

We tested our attack against the hash table stored in Linux' Netfilter (Filter,), which is the Linux IP firewall. The current Netfilter release is used in the 2.4 and 2.6 Linux kernels. Netfilter contains a stateful packet filtering module called *ip conntrack* which keeps state for each connection. Prior to the work of

(Crosby and Wallach, 2003), users of Netfilter complained already in July 2002 of a server slowdown that was attributed to a poor choice of hash function. In response to this issue, the developers of Netfilter switched their hash function to the Jenkins' hash (Jenkins, 1997), and additionally included a random secret value, known only to the server, as a parameter to the hash function. Thus, the Netfilter hash is protected against the basic attack of (Crosby and Wallach, 2003).

As it is used in Netfilter, the Jenkins' hash receives 4 parameters, each of 32 bits in length as follows: (i) the packet's source IP address, (ii) the packet's destination IP address XORed with the connection protocol number, (iii) a concatenation of the source and destination ports, and (iv) the secret random value known only to the *ip conntrack* module.

Rather than analyzing the uniformity of the bit-mixing in the Jenkins' hash (thus finding hash collisions), we consider the Jenkins' hash as a "black-box" which receives the above 4 parameters for each connection and returns a 32-bit output modulo the number of hash buckets. Our attack is based on creating packets that cause enough bucket collisions to achieve a recognizable slowdown.

The default conntrack hash table size is 2^{13} buckets. It is worth noting that the Linux developers recommend that a server used only as a firewall should increase the size of the hash table. Furthermore, the Netfilter developers limit the total number of connections in the hash table to 8 times the number of buckets, giving a default maximum of 2^{16} connections.[3] This limited capacity is actually a security measure: the attacker cannot attack the hash table by exploding the memory. On the other hand, as we shall see, the capacity is not small enough to avoid a server performance degradation. Throughout this paper, we consider *ip conntrack* in its default settings.

3 ATTACK OVERVIEW

3.1 Attack Constraints

To conduct an efficient remote algorithmic complexity attack against a hash table, the following prerequisites must be met: (i) the hash function (except the secret value) must be known to the attacker, (ii) the attacker must be able to produce enough packets that fall into the same bucket, and (iii), the attacker must be able to deliver these packets to the victim's server.

For network devices that track TCP/UDP connections, the items to be hashed are usually defined by the 96-bit tuple $<$ *src addr, src port, dst addr, dst port* $>$. The IP addresses are 32 bits each, and each port is of 16 bits length. However, the attacker cannot manipulate all 96 bits to produce collisions. First, the destination must be fixed to contain the victim's IP address. Next, most servers do not have all their ports open. In fact, many only have as few as two or three open ports. Therefore, for a basic algorithmic complexity attack (Crosby and Wallach, 2003), the attacker can manipulate the 32-bit source address, the 16-bit source port, and a few choices for destination port, which we take as another 2 bits, giving a total of 50 bits. As a result, the attack relies heavily on source IP address spoofing.

However, when we are attempting to determine the secret value in a randomized hash table (in the information gathering stage), we need to compute the RTT so we can detect the server's slowdown — i.e., the attacker needs to receive the server's SYN-ACK packets. This implies that the attacker must place her true IP address in the source IP address field, otherwise the server's responses will not be routed back to her. At a first glance, this seems to create a serious difficulty for the attacker: she only has about 18 bits to manipulate (only the source port and the open destination ports). For example, the default number of buckets in Netfilter for a server with 1GB of RAM is 2^{13}. Assuming that the hash function distributes its input uniformly across the hash table, even if all 2^{18} possible packets are tried, the expected chain size is only 32, which is too small to achieve a considerable list traversal slowdown.

3.2 Producing a Noticeable Slowdown

As noted in Section 3.1, if the attacker wants to measure the RTT then the source and destination IP addresses, as well as most of the destination ports bits, are fixed. The key observation is that the attacker is interested in the return trip time of a connection sent *after* the bucket filled up. In other words, while the bucket is filling up, the adversary does not care about the server's replies. Our solution, then, is to create 2 classes of attack packets per candidate secret value. Class A consists of many packets, with spoofed source IP addresses, that all fall into the same bucket. Class B packets are a small set of packets, with the source containing the true IP address of the attacker, that fall into the *same* bucket as those of the Class A packets. Note that the attacker creates all Class A and

[3] In fact, the number of possible connections stored in the hash is 2^{15}. However, each connection and its reversed tuple gets stored in the hash table, giving us a hash table capacity of 2^{16}.

Class B packets, for every possible secret value, *in advance*.

During the attack, the adversary iterates over all possible secret values, sending for each random value, X_i, a "large enough" number of Class A packets, followed by a small number of Class B packets. The attacker only measures the RTT of the Class B packets. Sending this relatively small number of packets will not result in a denial of service, but will produce a detectable slowdown, which is sufficient for us to identify the correct secret X.

3.3 Modeling the Attack Viability

3.3.1 Distribution of the Longest Chain

Let n denote the number of connections inserted into the hash table of size m. For the attack to be viable, the attacker must be able to construct at least a few Class B packets. As a concrete example, we consider $m = 2^{13}$ hash buckets, as in Netfilter, and $n = 2^{18}$ possible Class B packets. To find the distribution of the longest chain in any bucket, we consider a cell that follows a binomial distribution with $\mu = n/m = 2^{18}/2^{13} = 32$. Let M be a random variable representing the length of the chain in some cell. We want to find K such that $\Pr(M > K) \geq 1/2$, which derives to:

$$\frac{1}{2} \leq \Pr(M > K) = 1 - \Pr(M \leq K) = 1 - [F(K)]^m \quad (1)$$

Where $F(K)$ is the resulting cumulative distribution function. Substituting m to be 8192, we obtain the condition:

$$[F(K)]^{8192} \leq 1/2 \quad (2)$$

The normal approximation to the binomial distribution (with the error complementary function) yields:

$$\frac{1}{2} \geq \left[\phi\left(\frac{K + \frac{1}{2} - \frac{n}{m}}{\sqrt{\frac{n(m-1)}{m^2}}}\right)\right]^{8192} = \quad (3)$$

$$\left[\phi\left(\frac{K + \frac{1}{2} - 32}{\sqrt{32\frac{8191}{8192}}}\right)\right]^{8192} = \left[\phi\left(\frac{K - 31.5}{5.6565}\right)\right]^{8192} \quad (4)$$

$$\phi\left(\frac{K - 31.5}{5.6565}\right) \leq 0.5^{(1/8192)} \approx 0.99992 \quad (5)$$

$$\frac{K - 31.5}{5.6565} \lesssim 3.891 \quad (6)$$

$$K \lesssim 53.5 \Rightarrow K \leq 54 \quad (7)$$

Thus, we see that with high probability we can create chains of length 54, even for an ideal hash function — this is more than enough for Class B packets.

3.3.2 Calculating the Number of Class A Packets

We also need to estimate the number of packets needed to be sent for each secret value, in the information gathering stage, for the attacker to achieve a recognizable slowdown. Assume that the normal RTT is T. Let $T_r(n)$ denote the RTT for Class B packets, after sending n Class A packets, assuming a secret value of r. We would like to achieve $T_r(n) > (1+\alpha)T$ if r is the correct secret value, for some fixed $\alpha > 0$.

Let t_0 to be the time to traverse from one node to another in a linked list (the *lookup time*) and let t_1 be twice the network propagation delay between the attacker and victim, i.e., the "network" part of the RTT. When all the connections are distributed uniformly throughout the hash table, we calculate the following expectation to receive a reply from the server after creating n connections: $T = t_0(n/m) + t_1$ since, on average, the servers needs to examine n/m tuples in the hash function. When all the connections fall into the same hash bucket, the time that it takes to receive a reply after sending n packets is: $T_r(n) = nt_0 + t_1$. To achieve $T_r(n) \geq (1+\alpha)T$ we need

$$(1+\alpha)[(\frac{n}{m}t_0 + t_1)] < nt_0 + t_1 \quad (8)$$

which implies that

$$\frac{n - \frac{n}{m}(1+\alpha)}{\alpha} > \frac{t_1}{t_0} \quad (9)$$

For simplicity we set $\alpha = 1$ (for a slowdown of 2). Then for $m = 2^{13}$ we have

$$n(1 - \frac{2}{2^{13}}) > \frac{t_1}{t_0} \quad (10)$$

As a crude estimate, if we set the ratio t_1/t_0 to be 1000, we see that $n \approx 1000$ Class A packets would suffice to produce a factor of 2 slowdown in the RTT when we test the correct secret value.

Such a ratio is reasonable when the propagation delay is, e.g., $\approx 1ms$ (an attacker is stationed just a few hops away from the victim), and the lookup time is $\approx 1\mu s$. Clearly, as the noise in the network increases, or the distance between the attacker and the victim increases, t_1 increases which means that we need to create more connections to recognize a slowdown. Likewise, a faster victim machine would require more connections.

Note that in reality we don't really require a slowdown of 2, all we need is that the RTT for the correct secret value should be largest among the RTTs computed for all possible candidates.

4 ATTACK IMPLEMENTATION AGAINST NETFILTER

Before we implemented our attack, we conducted an extensive simulation study using NS2 (McCanne and Floyd,). We omit the details due to space constraints.

4.1 Implementation Setup

We implemented this attack by running a real-life experiment between 2 machines sitting on the same network switch. A computer containing a 3.4GHz Intel Pentium 4 CPU with 1GB RAM, running Fedora Core 4 distribution with a Linux kernel version 2.16.14, served as the attacked server. The victim machine had only 3 open ports and an installed Netfilter with *ip conntrack* version 2.3. We changed the Netfilter module, *ip conntrack*, so that it can receive the size of the random value in bits as a parameter to the module. The attacker machine, composed of an Intel Pentium 4 2.4GHz and 512MB RAM, ran a Red Hat 9 Linux distribution with a 2.4.2 kernel.

We wrote a simple C program to generate 40 Class B tuples for each candidate secret value (with the attacker's source IP address) which all enter the same hash bucket, assuming that the current candidate secret value is the one used in the server. With hindsight, 10 class B would have been sufficient.

After generating 40 Class B tuples, the program then generates another 1500 Class A tuples that fall into that same bucket, but with fake source IP address. For the forged addresses, we chose to use the 10.X.X.X address space (2^{24} choices) to avoid the true IP addresses from sending RST packets which would cause a connection to be purged from the hash table.[4] Constructing the tuples with these forged IP addresses, leaves us with 42 bits of freedom, still enough to cause hundreds of bucket collisions. Once we generated all the tuples for the experiment (an offline computation), we can test sending packets in bursts of different sizes between the attacker and the victim in order to recognize a slowdown.

To inject and capture TCP packets we used the packit (Bounds, 2003) application with minor modifications of our own to accommodate the experiment.

[4]When we used totally random source addresses, the victim's SYN-ACK packets were routed towards the Internet. The campus PIX firewall would trap them in its egress filtering mode, identify them as "out-of-state", and send a spoofed TCP-RST back to the victim, causing the victim machine to tear down the half open connection and thwart the attack. This counter-measure would not affect the attack in a real scenario when the attacker is outside the perimeter because the border firewall would see both the attack SYN and the victim's SYN-ACK.

We iterated over all possible secret values, sending first a variable-length burst of Class A packets to the server, and then 10 Class B packets (with the real source IP address). We ran this experiment in bursts of 200, 500, 600, 750, 900, 1000, 1200, 1350, 1500 packets and repeated this sequence twenty times.

For each burst sent for each secret value, we calculated the average RTT over the Class B packets and performed our statistical tests on this data. However, we noticed that occasionally the average RTT of a certain value was unreasonably high due to transient network congestion on the switch. Thus, we considered these abnormally high times as *outliers* which we excluded from our statistical tests. The threshold to consider a data point as an outlier was set to $1500\mu s$, taken as 10 times the average RTT between the 2 machines under normal circumstances. Note though that on some of the tests, also the correct secret value showed up as an outlier and so was dropped from the statistics as well. We calculated the fraction of times the highest average RTT was detected for the actual secret value, the fraction of times the true secret value RTT was in the top-5, and when it was in the top-10.

We ran this experiment twice: The first experiment set the random value's size to 13 bits while the second set the size to 14 bits.

4.2 Experiments Results

4.2.1 13-bit Secret Values

In the first experiment the random value had 13 bits. The offline generation of packets lasted about 54 hours and produced 350MB of file space. On average, it took 83,000 different combinations to find 40 Class B tuples for each random value when hashing with the attacker's real source IP addresses, and it took 12,000,000 different combinations to find 1500 Class A tuples that collide into the same bucket as the Class B tuples for the same secret value. The online slowdown recognition experiment lasted 15 days.

Note that a slow information gathering stage is not unreasonable. In fact, the attacker may choose to use a low transmission rate to avoid being detected as an old-fashioned SYN-flood attack. In our experiment the overall transmission rate was 33 Kbps (1,048 SYN packets per second) and with short bursts of 480 Kbps.

Figure 1 presents the statistical results. Out of all the tests, 31 outliers were removed from the data. Out of these, 6 of the outliers were actually the real random value. One can see in the graph that as the number of packets sent per burst increases, the high

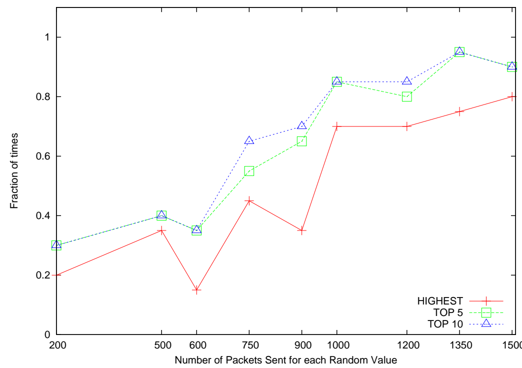

Figure 1: Fraction of times in which the correct secret value caused the highest RTT, was in the top-5 RTTs, or the top-10 RTTs, as a function of the number of Class A packets sent. The secret value is of 13 bits.

RTT values are the result of finding the actual random value. However, even sending 200 packets per burst gives a success rate of 20% for having the highest RTT value belong to the actual secret value. For a burst of 750 packets per candidate secret value, the correct value's average RTT was one of the ten highest RTT values on more than half of the tests. The drops in the graph for the top-5 and top-10 highest values were caused by the removal of the outliers as presented above, i.e., by poor transient network conditions. The graph shows that with a burst size of 1000, the attacker guessed the true secret value almost always. According to these results, it can be safely said that 1000 packets are enough to detect the correct secret value during the information gathering stage, when the secret value is 13 bits long. These results closely match those we previously calculated in Section 3.3.2. Figure 2 shows the differences in RTT values for each candidate value taken in a single run with a burst size of 500. The figure on the top shows that the highest RTT is the one calculated for the actual secret value (which is set to be 5766). However, the bottom figure is also a single run with a burst size of 500. In this case, the RTT of the true secret value was not even one of the top-10 highest received RTTs.

4.2.2 14-bit Secret Values

For the experiment when the secret value is 14 bits long, the offline pre-processing time and space is multiplied by a factor of 2: the tuple generation lasted about 104 hours on the same computer and produced 700MB of space. The number of combinations to create tuples that fall into the same bucket, both when forging and not forging the source IP address are similar to the results received when the random value is 13 bits long: on average over all the random values,

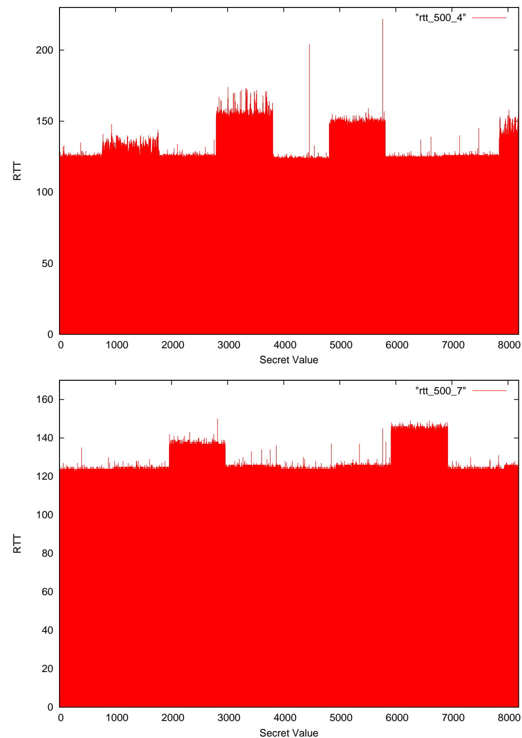

Figure 2: Comparison when the highest RTT belongs to the actual random value (5766) (top), and when the RTT of the random value was not even in the top 10 highest RTTs when the network was generally slower (bottom). A burst of 500 was sent for both these tests. RTTs are specified in microseconds.

83,000 combinations were tried to create 40 Class B tuples, and 12,000,000 combinations were tried to create 1500 Class A tuples that all fall into the same bucket. The experiment for the 14 bit random value length lasted about 31 days.

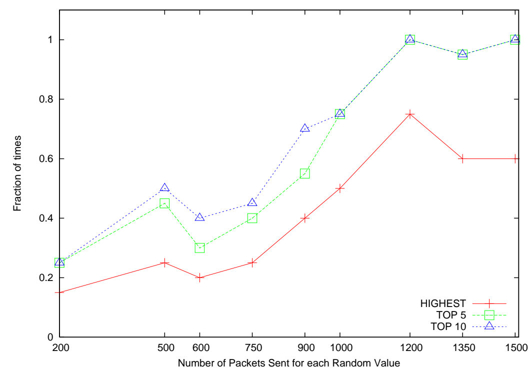

Figure 3: Fraction of times in which the correct secret value caused the highest RTT, was in the top-5 RTTs, or the top-10 RTTs, as a function of the number of Class A packets sent. The secret value is of 14 bits.

The experiment results are shown in Figure 3. The number of outliers in this experiment was higher, where 58 results were removed. Out of these values, 6 outliers were caused by the extremely high RTT values received for the real random value. The drop in the top-5 and top-10 success rate in the figure for the burst size of 1350 is due to the outlier being the actual secret value. The figure is very similar to Figure 1: as the burst size grows, a slowdown in the RTT values is almost always due to finding the correct secret value. The figure shows that as few as 500 packets suffice to recognize the correct secret value in the top-5 with $\approx 50\%$ success.

5 CONCLUSIONS AND FUTURE WORK

We have demonstrated that a remote algorithmic complexity attack, against randomized hash tables, is possible if the secret value is chosen from a small enough space. More secret bits cause more effort, time and space to be consumed in the *information gathering* stage. Thus, it seems that a random value of 32 bits would render this attack impractical with today's technology. Note though that in this paper the attacker iterates over all possible random values in a brute-force manner, searching for bucket collisions. However, the search space may be limited to a smaller subset of random numbers by taking advantage of the vulnerabilities in the Linux Random Number Generator as suggested in (Gutterman et al., 2006). This might lead to a feasible attack against a server with a longer secret value.

The Linux Routing Table cache which uses a hash table, has also updated its hash function as a countermeasure against the algorithmic complexity attack with Linux version 2.4.2. In this patch, the routing table cache also uses a random value as a parameter to the hash function, but in order to increase the security, this key is changed every 10 minutes. Since our experiments show that when the random value is 13 bits long, testing all 8192 possibilities with 500 packet bursts takes about 1 hour, this additional measure is indeed helpful. However, changing the secret value is not always easy: Doing so on a firewall like Netfilter will potentially break existing connections since future packets will be hashed to a different bucket and not find the connection's state.

REFERENCES

Boneh, D. and Brumley, D. (2003). Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*.

Bounds, D. (2003). packit v1.0. `http://www.obtuse.net/software/packit/`.

Crosby, S. and Wallach, D. (August 2003). Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44.

Dean, D. and Stubblefield, A. (Aug. 2001). Using client puzzles to protect TLS. In *Annual USENIX Security Symposium*, page 178, Washington, D.C., USA.

Filter. Linux netfilter. `http://www.netfilter.org/`.

Gal, A., Probst, C., and Franz, M. (2004). Complexity-based denial of service attacks on mobile-code systems. Technical Report 04-09, School of Information and Computer Science, University of California, Irvine.

Gal, A., Probst, C., and Franz, M. (2005). Average case vs. worst case margins of safety in system design. In *Proceedings of the 2005 New Security Paradigms Workshop (NSPW 2005)*, Lake Arrowhead, CA, USA.

Garfinkel, S. (1996). Script for a king. *HotWired Packet*.

Gutterman, Z., Pinkas, B., and Reinman, T. (2006). Analysis of the linux random number generator. In *IEEE Symposium on Security and Privacy*, Berkeley/Oakland, CA, USA.

Jenkins, B. (1997). Jenkins' hash. `http://burtleburtle.net/bob/hash/doobs.html`.

Kohno, T., Broido, A., and Claffy, K. (2005). Remote physical device fingerprinting. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA.

Kuzmanovic, A. and Knightly, E. (2003). Low-rate TCP-targeted denial of service attacks (the shrew vs. the mice and elephants). In *Proc. Sigcomm*.

McCanne, S. and Floyd, S. ns network simulator. `http://www.isi.edu/nsnam/ns/`.

McIlroy, M. D. (1999). A killer adversary for quicksort. *Softw., Pract. Exper.*, 29(4):341–344.

Needham, R. M. (1993). Denial of service. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 151–153, FairFax, VA, USA.

Paxson, V. (1999). Bro: a system for detecting network intruders in real-time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23–24):2435–2463.

RFC4418. Umac: Message authentication code using universal hashing. `http://www.rfc-archive.org/getrfc.php?rfc=4418`.

Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh., D. (2004). On the effectiveness of address space randomization. In *ACM Conf. Computer and Communications Security (CCS)*, pages 298–307.

SYN flood (1996). SYN-flooding attacks. `http://www.cert.org/advisories/CA-1996-21.html`.