Detection of Cache Pollution Attacks Using Randomness Checks

Hyundo Park
Bell Labs
Murray Hill, NJ, USA
hyundo.park@alcatel-lucent.com

Indra Widjaja
Bell Labs
Murray Hill, NJ, USA
iwidjaja@research.bell-labs.com

Heejo Lee
Korea University
Seoul, Korea
heejo@korea.ac.kr

***Abstract*— The Internet plays an increasing role in content dissemination as user-generated contents have exploded recently. Cache servers have been deployed to bypass bottlenecks in the network so that contents can be delivered to end users more efficiently. With caches becoming more embedded in the networks, emerging threats follow naturally. A cache pollution attack is one of the most serious threats on caching networks including the current Internet and emerging caching networks such as Content Centric Networking (CCN). In this paper, we propose a detection approach against cache pollution attacks using randomness checks of a matrix. We apply an effective filtering approach and a statistical sequential analysis for detecting low-rate attacks. The results of our experiments show that our approach can detect a cache pollution attack with attack rate of only a few percent of the overall rate.**

Keyword: caching network, cache pollution attack, matrix operation, randomness checks

I. Introduction

Guaranteeing the integrity of cache servers against attacks is extremely important for today's and future Internet as these servers are expected to provide high service levels continuously to end users. However, with proliferation of web caching and media caching, attacks on such systems will become increasingly more common. Recently, researchers have begun to look at the robustness of Internet cache server under cache pollution attacks [1], [2]. They show that the performance of a cache server can be seriously degraded by a cache pollution attack. It is well known that many popular contents are usually requested by legitimate users and the distribution of their requests typically follows a Zipf-like distribution [3]. To degrade caching service, the attackers can attempt to violate the locality of contents in the cache with unpopular contents so that popular contents requested by legitimate users are evicted.

Fig. 1 shows how the cache hit ratio can be degraded from a cache pollution attack where the attackers request content objects according to a uniform distribution. This degradation can be serious when caches are pervasive and adopted in every network node. Indeed, a recent proposed network architecture, called Content-Centric Networking (CCN), implements a cache in each router [4]. Therefore, it becomes more crucial in the future to have an effective approach to deal with cache pollution attacks.

A cache pollution attack may be classified into two types: locality-disruption and false-locality [1]. In locality-disruption attack, the attackers continuously generate requests for new unpopular contents, thus disrupting the content locality of a cache. In false-locality attack, the attackers repeatedly request the same set of contents that are not requested by legitimate users, thus creating a false locality in a cache. These two attack types are intended to degrade the cache performance. In false-locality attack, an attacker must know the distribution of popular contents by legitimate users. Inferring the total distribution of popular contents by legitimate users is generally not realistic as popular content distribution is changing as a function of time and location of a cache. In contrast, for locality-disruption attack, an attacker needs only request contents with the uniform distribution to the cache server. It means that the attacker, even without knowledge of popularity information, can severely damage the effectiveness of a cache server using locality-disruption attack.

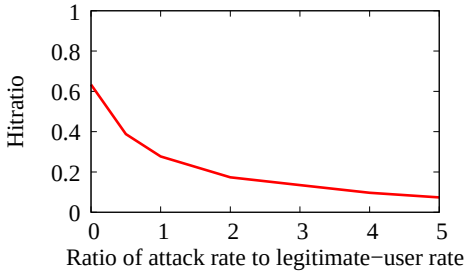


Fig. 1. Hit ratio degrades under attack.

In this paper, we focus on the locality disruption attack. We propose a real-time scheme that can detect low-rate attacks on caches. Previous works deal with high-rate attacks in the Internet without caching taken into consideration [5], [6]. In this paper, we find that it is more effective for the detection to instrument the measurement inside the cache rather than at a link. We use the rank value of a matrix to check its randomness as a proxy of an attack. The rank value can be easily calculated by Gaussian elimination. The detection becomes effective when filtering is done with certain XOR and AND operators on two or more matrices to remove requests from legitimate users so that requests from the attackers become more apparent. Under a very low attack rate, the rank value may only change in a small way. We adopt a statistical sequential analysis in our approach to deal with low-rate attacks. The results of our experiments show that our approach can detect low-rate attacks

This work was partially supported by Seoul City R&BD program WR080951

perfectly (i.e., with true positive ratio of 1 and false positive ratio of 0) even when the attack rate is 2% of the legitimate-user rate.

The rest of this paper is organized as follows. In section II, we present an overview of our approach. In section III, we present the detail of our mechanism and the results of our experiments. In section IV, we evaluate the effectiveness of our approach in detecting low-rate attacks. Finally, we conclude this paper in Section V.

II. Overview of The Proposed Approach

A. Randomness Checks

It is well known that objects requested by legitimate users follow Zipf-like distribution. In locality-disruption attack, the attackers continuously request unpopular objects, thus forming a (near) uniform distribution. We propose to use a rank value of a matrix to check the distribution of objects in a cache and demonstrate its effectiveness.

Since a random binary matrix holds a high rank value [7], the rank value of a matrix has been widely used for testing the quality of a random number generator such as the Diehard battery tests [8]. The rank value of a binary matrix is the count of non-zero rows after applying Gaussian elimination. In the case of $n \times m$ binary matrix, the probability that the matrix has rank r is

$$2^{r(n+m-r)-nm} \prod_{i=0}^{r-1} \frac{(1-2^{i-n})(1-2^{i-m})}{(1-2^{i-r})} \quad (1)$$

where $1 \leq r \leq min\{n, m\}$ [7]. In our approach, we use the rank value of a matrix to detect a change of the distribution of objects. To do this, we first use a function to map each object[1] to an entry in a matrix, $M = (m_{i,j})$, as follows:

$$f : X \longrightarrow Y \quad (2)$$

where X is the object name space and $Y = \{(i,j), 1 \leq i \leq n, 1 \leq j \leq m\}$. The entry $m_{i,j}$ is then set to 1 if an object name is mapped to (i,j) and 0 otherwise.

To deal with low-rate attack, we next device a matrix operation with contiguous matrices, M_{t-2}, M_{t-1} and M_t, as follows:

$$M_{XOR}(t) = M_{t-1} \oplus M_t \quad (3)$$

$$M_t{}' = M_{XOR}(t) \oplus (M_{XOR}(t) \bullet M_{t-2}) \quad (4)$$

where t is the time instant when the matrix operation is performed, $\oplus$ is an exclusive-OR operation between each corresponding entry of two matrices and $\bullet$ is an AND operation. The time interval between two time instances is one *time unit*. The motivation for using the matrix operation Eq. 3 and Eq. 4 is that it offers a powerful technique to deal with low-rate attack. In particular, popular objects that are repeatedly requested by legitimate users are easily removed by the matrix operation as they are likely to map to the same indices in contiguous time units. On the other hand, the objects requested by the attackers are likely to map to different indices in different time units.

After applying the matrix operation, we then obtain the rank value r_t of matrix $M_t{}'$ using Gaussian elimination. We can use r_t as an indicator of an attack.

[1]The mapping function uses an object name and we use the term object and object name interchangeably for mapping.

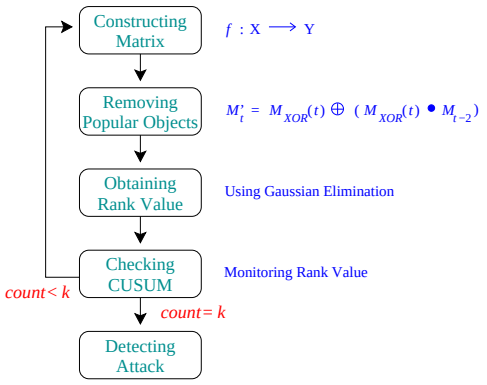


Fig. 2. Flowchart of our approach.

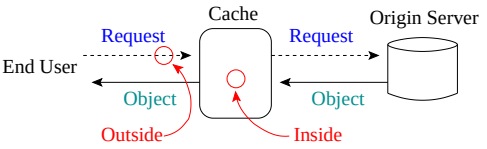


Fig. 3. Locations where instruments are placed.

B. Statistical Sequential Analysis

In order to detect an attack accurately and timely, we adopt the cumulative sum (CUSUM) algorithm in our approach [9]. CUSUM has also been used for change point detection and applied in anomaly detection [10]–[12]. If $\overline{\mu}_{t-1}$ is the mean of the rank value estimated from measurements prior to t, $\overline{\mu}_t$ can be obtained using an exponentially weighted moving average (EWMA) as follows:

$$\overline{\mu}_t = \beta \overline{\mu}_{t-1} + (1-\beta) r_t \quad (5)$$

where β is the EMWA factor. CUSUM can detect a small deviation of the mean effectively. It is generally defined as follows:

$$\begin{cases} g_t = [r_t + g_{t-1} - \overline{\mu}_t]^+ \\ g_0 = 0 \end{cases} \quad (6)$$

where $[x]^+ = x$, $x > 0$ and $[x]^+ = 0$, otherwise, and r_t is the observed original rank value at time t. An alarm is signaled when g_t exceed a threshold h for k consecutive time units.

C. Summary of the approach

Fig. 2 summarizes our overall approach for detecting a cache pollution attack. The sequence of the operations in the flowchart is executed at every time unit and is halted when an attack is detected and an alarmed is signaled.

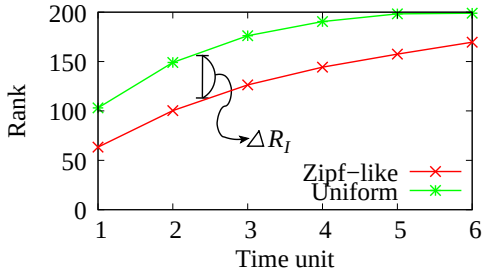


Fig. 4. Rank values estimated from the inside of a cache.

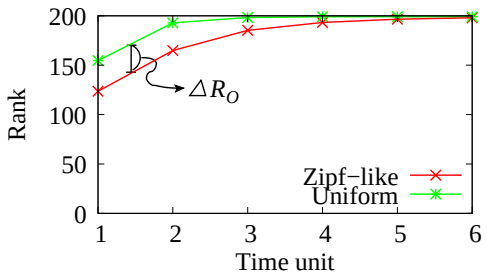


Fig. 5. Rank values estimated from the outside of a cache.

III. Details of the Mechanism

In this section, we describe the details of how the major components in the flowchart are designed or dimensioned. We also discuss our experiments and describe the results.

A. Location Issue

Object names to be mapped to a matrix can be done in two ways. Object names in user requests can be measured at the input to a cache proxy server (outside) *during each time unit*. Alternatively, they can be measured from the cache itself (inside) *at the end of each time unit* (see Fig. 3). We evaluate how the rank values differ when object names are measured from inside and outside of the cache proxy server. Let S_O denote the total number of objects and S_C denote the average number of objects in the cache. We use an $n \times n$ matrix in the matrix operation. For illustration, we assume that $S_O = 10,000,000$, $S_C = 40,000$, $n = 200$, and that one time unit corresponds to 500 requests for objects. Fig. 4 plots the rank values as a function of time units. ΔR_I is the difference of the rank values between a Zip-like distribution and a uniform distribution where objects are measured from inside a cache proxy server. By comparison, ΔR_O in Fig. 5 is the difference of the rank values measured outside of the server.

When objects are measured from the outside (see Fig. 5), we observe many unpopular objects that are requested by legitimate users in one time unit even if there is no attack. This weakens the detection since unpopular objects tend to increase the rank value. Measured from inside (see Fig. 4), the filtering effect by the cache can relieve this problem since the cache replacement algorithm evicts unpopular objects from the cache. This allows us to distinguish the rank values of objects requested by the legitimate users and those requested by the attackers more clearly; that is, $\Delta R_I > \Delta R_O$. In the following, the results are obtained from measurements inside the cache server.

B. Mapping function

To map objects in the cache to a matrix, we need to obtain an index value (i, j) of a matrix corresponding to each object name. The function to obtain an index value of a matrix is a 'Mapping function'. In general, the name of an object is composed of a string. A mapping function should have three desirable properties as follows:

First, it should be easy to compute the index value for any given object. If the computation of the mapping function is complicated, it may cause cache server overload.

Second, it should be difficult to find an object that maps to a given index value. If this property is violated, the attackers that seemingly request many unpopular objects can end up with the same index value.

Third, it should be difficult to modify the object name without the index value being changed. If this property is violated, the rank value will be reduced and this makes the detection less effective.

To satisfy these three properties, we adopt a cryptographic hash function to map an object name to an index value [13]. The mapping function is defined as follows:

$$\begin{cases} i = Hash_1(c) \mod n \\ j = Hash_2(c) \mod m \end{cases} \tag{7}$$

where (i, j) is the index of the matrix corresponding to object name c and $Hash_1$ and $Hash_2$ are cryptographic hash functions. In our evaluations of this paper, we used SHA-1 as a hash function.

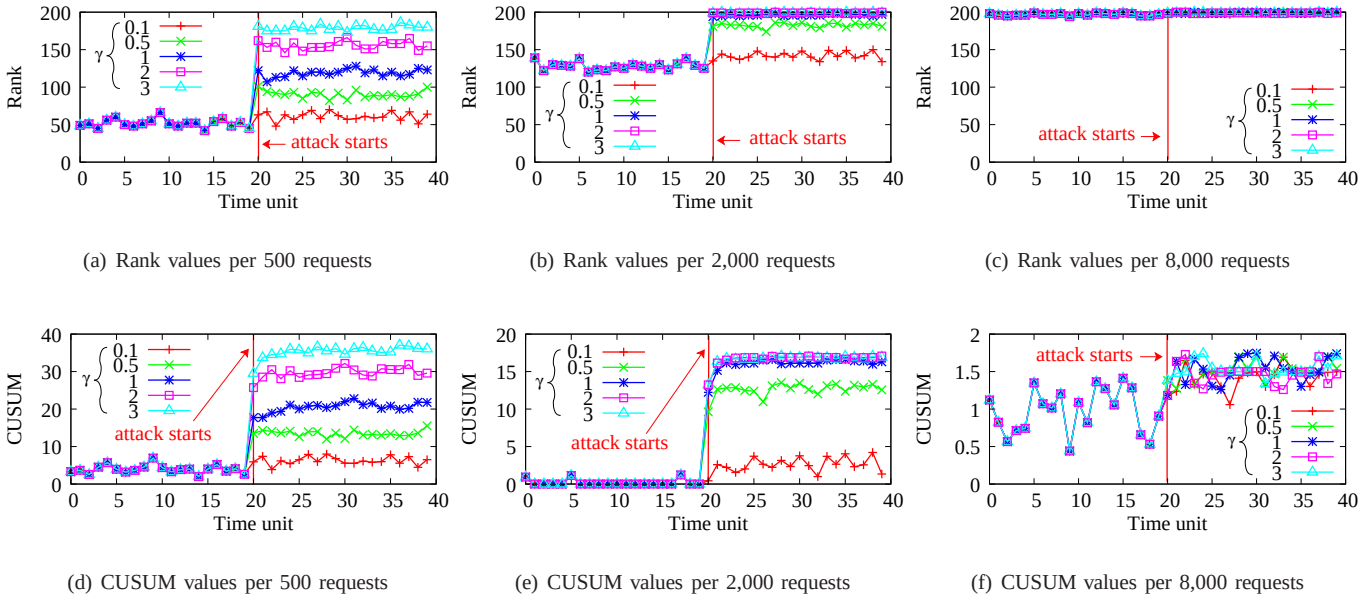
C. Matrix size

Another important issue to consider is matrix size. Let M be a $n \times n$ binary matrix. It has been determined that if $n < 10$, then we cannot check the randomness of a matrix [7].

Since we map the objects in the cache to a matrix at each time instant, the matrix size in our case depends on the average number of objects stored in a cache, S_C. If we choose a very large value of n such that $n^2 \gg S_C$, many locations in the matrix will have zero entries. On the contrary, if $n^2 \ll S_C$, there will be considerable collisions in the matrix, where a collision occurs when two or more distinct objects map to the same entry in the matrix. We choose the matrix size

$$n \simeq [\sqrt{S_C}]. \tag{8}$$

For example, if the average file size is 1M Byte and the size of the cache server is 1T Byte, then n will be roughly $1,000$.



(a) Rank values per 500 requests (b) Rank values per 2,000 requests (c) Rank values per 8,000 requests

(d) CUSUM values per 500 requests (e) CUSUM values per 2,000 requests (f) CUSUM values per 8,000 requests

Fig. 7. Rank values and CUSUM values of various time units

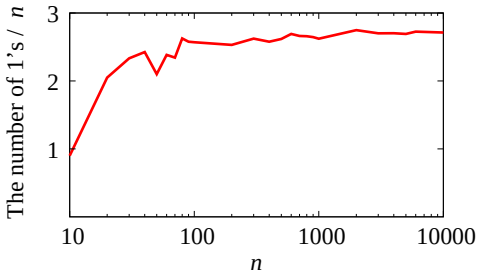


Fig. 6. Ratio of average number of 1's to n versus n.

D. Time unit

We now describe how the time unit should be chosen. To have the time unit decoupled from request arrival rate, we use the number of requests to determine the time unit instead of actual time duration. For example, a good time unit in Fig. 4 would be around 1500-2000 requests where the rank value is (near) full when the requests are from attackers. If the time unit is too large, the rank value will also be close to full even if the requests are from legitimate users.

To explore the time unit, we perform an experiment to determine the required number of distinct objects requested by the attackers to achieve full rank with different matrix sizes. The number of distinct objects requested can be measured by the number of 1's in the matrix when requests are from attackers as collisions are rare. Fig. 6 plots the ratio of the minimum number of 1's to n versus n to achieve full rank with an $n \times n$ matrix. From the figure, we conclude that a reasonable choice for the number of requests, or the time unit, should be about $3n$. We also need to make some additional adjustment when objects are measured in a cache.

E. Detection

When a cache pollution attack is triggered on the cache proxy server, the distribution of objects in a cache will change and the rank value will increase as a result. We use the CUSUM algorithm based on the rank value, as described in Sec. II-B, to detect a cache pollution attack.

Previous work uses the rank value to detect attacks when the rank value exceeds a threshold that is set $n-4$ [5]. Under high-rate attacks, such as worm or DDoS attacks, the rank value will always go over the threshold even if we select a short time period such as 10 sec. When a low-rate attack is triggered toward the cache server, the rank value will not go over the threshold, but instead changes from a low value to some value higher depending on the rate of the attack. By applying CUSUM, we can detect low-rate attacks.

In Fig. 7, we explore the results obtained via the rank value and CUSUM. We also compare the effectiveness of the detection mechanisms with different time units. We assume $S_O = 10,000,000$, $S_C = 40,000$ and $n = 200$. Let γ denote the ratio of the attack rate to the legitimate-user rate. A cache pollution attack is triggered on the cache server with $\gamma = 0.1, 0.5, 1, 2, 3$ after $10,000$ requests (in 7(a) and 7(d)), $40,000$ requests (in 7(b) and 7(e)), $160,000$ requests (in 7(c) and 7(f)). In the CUSUM algorithm the value of EWMA factor, β is set 0.25. We also set $h = 0$ and $k = 5$. The value of h may be set in practice based on observation of average CUSUM value under normal situation (no attack).

Using 2000 requests per time unit, observe that we can detect attacks with attack ratio ($\gamma = 1, 2, 3$) when the threshold is set to $n-4$ in Fig. 7(b), and we can detect all attacks in Fig. 7(e). If the time unit is too short, both rank value and CUSUM will become ineffective since the rank value cannot detect any attack (as depicted in Fig. 7(a)) while CUSUM misdetects in the normal situation (as depicted in Fig. 7(d)).

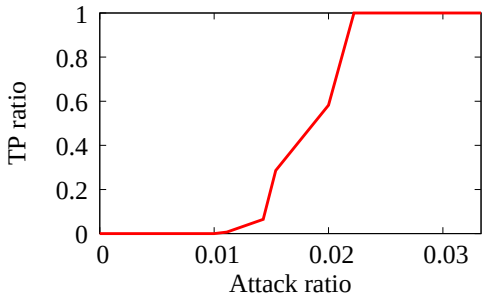


Fig. 8. True positive (TP) ratio of a locality disruption attack detection

		Detection	
		Attack	No attack
Truth	No attack	False positive α (0)	True negative $1-\alpha$ (1)
	attack	True positive β (1 if $\gamma > 0.02$)	False negative $1-\beta$ (0 if $\gamma > 0.02$)

TABLE I
DISTRIBUTION OF THE TYPES BY RANK

If the time unit is too long, both rank value and CUSUM will also become ineffective as both will misdetect in the normal situation (as depicted in Fig. 7(c) and Fig. 7(f)). When the time unit is chosen properly, CUSUM can detect low-rate attacks while the rank value can only deal with high-rate attacks.

IV. Effectiveness of our approach

In this section, we evaluate the effectiveness of our detection approach under locality-disruption attack. In [1], the authors measure the average life-time of all cached files to detect locality-disruption attack. To do that, they record the entry time of each cached files and compute the average duration for all files in the cache periodically. When the average duration is very low, their approach warns a presence of locality-disruption attack.

Since their approach checks the average life-time of all cached files, the average life-time will only change insignificantly if the attack ratio is too small. In addition, their approach is based on the assumption that the number of attackers is much smaller than the number of legitimate users, but the number of requests per attacker is much larger than the number of requests per legitimate user. Clearly their assumption will be violated when an attacker spoofs his/her own IP address or uses Botnet since their approach will not be able to detect the attackers. By comparison, since our approach utilizes the distribution of objects in the cache and further filters the objects requested by legitimate users by means a matrix operation, our approach can effectively detect low-rate attacks. Additionally, our approach can detect an attack even if an attacker spoofs his/her IP address or uses Botnet.

When the ratio of the attack rate to the legitimate-user rate is $\gamma = 0.3$, their approach correctly detects around 60 percent of the attacks; that is, the true positive ratio (TP ratio) is 0.6. By comparison, our approach can correctly detect all attacks (TP ratio is 1) when $\gamma = 0.3$. Fig. 8 plots the TP ratio versus attack ratio, γ, in our approach. We use $S_O = 10,000,000$, $S_C = 40,000$, $n = 200$, a time unit of $2,000$ requests and $k = 5$. The results of the TP ratio is obtained with $10,000$ times of repeated evaluations. As shown in Fig. 8, our approach can correctly detect all attacks when the attack ratio is $\gamma \approx 0.02$. We observed that the false negative (FN) ratio is 0. Note that our approach only fails to detect an attack when $\gamma \approx 0.01$. When we change k to 1, the FN ratio of our result is 0.007 (falsely detect an attack when there is no attack). In addtion to the FN ratio, as shown table I, we also observed that the false positive (FP) ratio, α, is always 0 if we choose a good time unit (see Sec. III-D). Furthermore, since our approach uses the number of requests as a time unit, the rank of normal requests does not change even when the cache is under the flash event (FE). Under FE, the number of requests from normal users are suddenly increased.

V. Conclusion

In this paper, we have proposed a mechanism to detect a cache pollution attack using randomness checks of the distribution of content objects requested by legitimate users as well as by the attackers. To detect low-rate attack, we adopted the CUSUM technique in our approach. Our experiments have shown that the proposed mechanism can detect a cache pollution attack even when the attack rate is very low. Our method significantly improved the previous approach as it can still effectively detect cache pollution attack while the attack rate is an order of magnitude less than that in the previous approach.

References

[1] Y. Gao, L. Deng, A. Kuzmanovic, and Y. Chen, "Internet Cache Pollution Attacks and Countermeasures," in *IEEE ICNP*, pp. 54–64, 2006.

[2] V. Manivel, M. Ahamad, and H. Venkateswaran, "Attack Resistant Cache Replacement for Survivale Services," in *SSRS '03: Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems*, pp. 64–71, 2003.

[3] L. Breslau, P. Cue, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," in *INFOCOM*, pp. 126–134, 1999.

[4] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking Named Content," in *ACM CoNEXT*, pp. 1–12, 2009.

[5] H. Park, P. Li, D. Gao, H. Lee, and R. H. Deng, "Distinguishing between FE and DDoS using Randomness Checks," in *Information Security Conference(ISC), LNCS*, vol. 5222, pp. 131–145, sep 2008.

[6] H. Park, H. Kim, and H. Lee, "Is Early Warning of an Imminent Worm Epidemic Possible?," in *Network, IEEE*, vol. 23, pp. 14–20, sep 2009.

[7] G. Marsaglia and L. H. Tsay, "Matrices and the Structure of Random Number Sequences," *Linear Algebra and its Applications*, vol. 67, pp. 147–156, 1985.

[8] G. Marsaglia, "Diehard: A Battery of Tests of Randomness," 1996. `http://wtat.fsu.edu/pub/diehard`.

[9] M. Basseville and V. Nikiforov, *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, 1993.

[10] V. A. Siris and R. Papagalou, "Application of Anomaly Detection Algorithms for Detecting SYN Flooding Attacks," *Computer and Communications*, vol. 29, pp. 1433–1442, 2006.

[11] H. Liu and M. S. Kim, "Real-time Detection of Stealthy DDoS Attacks using Time-series Decomposition," in *IEEE International Conference of Communications*, 2010.

[12] A. Dainotti, A. Pescape, and G. Ventre, "Wavelet-based Detection of DoS Attacks," *Globecom 2006*, vol. 25, pp. 1452–1457, nov 2006.

[13] P. Rogaway and T. Shrimpton, "Cryptographic Hash-function Basics: Definitions, Implications, and Separations for Preimage Resistance, Seconde-Preimage Resistance and Collision Resistance," *Fast Software Encryption (FSE), LNCS*, vol. 3017, pp. 371–388, 2004.