

HellaSec

Technical report

Bouncer: Provably Protecting Servers From High-Density Resource Consumption DoS Attacks

February, 2012

Michael N. Gagnon,
HellaSec

mike@hellasec.com

Stephen Chong,
Harvard University

chong@seas.harvard.edu

Bouncer: Provably Protecting Servers From High-Density Resource-Consumption DoS Attacks

Michael N. Gagnon
HellaSec
Mountain View, CA 94043
mike@hellasec.com

Stephen Chong
School of Engineering and Applied Sciences
Harvard University
Cambridge, MA 02138
chong@seas.harvard.edu

Abstract—The typical DoS attack is a *high-volume / low-density* attack whereby an adversary consumes victim resources through a *large* volume of malicious actions, where each action individually consumes only a *small* portion of resources. A more threatening form of attack is a *high-density* attack, which increases *total* resource consumption by increasing the resource consumption of *individual* malicious actions.

In this work we investigate a mechanism that can protect servers from high-density attacks. The mechanism works by evicting “greedy” requests when the server becomes overloaded. Our formal theoretical analysis shows that the technique provides provable graceful degradation in the face of high-density attacks. Experimental results using Wikipedia as a case study demonstrate that the mechanism is practical and effective in practice.

I. INTRODUCTION

Denial-of-service (DoS) attacks continue to trouble the Internet. Perhaps the most common attacks are *resource-consumption* DoS attacks, whereby a malicious adversary consumes a disproportionate amount of the victim’s resources—preventing legitimate users from using those resources. These attacks target a variety of resources (such as CPU, memory, and network bandwidth) and use a variety of means to intensify the attack (such as increasing the rate of requests or exploiting protocol idiosyncrasies). In this work we are concerned with a particularly potent form of resource-consumption DoS attack, which we term high-density resource-consumption DoS attacks.

A *high-density resource-consumption* attack increases the overall *mass* of the attack (the amount of resources consumed) by increasing the resource-consumption of individual malicious actions—as opposed to merely increasing the *volume* of malicious actions (the number of actions). To illustrate, consider a Wiki web-application. One can attempt a *low-density / high-volume* DoS by sending many page-view requests to the server at a high rate. If the server is not provisioned to handle that many requests, then it will become overloaded and will be less able to respond to the legitimate requests buried in the sea of malicious requests. In contrast, one can attempt a high-density attack by sending *high-density requests*—requests that individually consume

more resources. For example, rather than sending page-view requests one can request “diffs” (descriptions of the differences between pages). Since calculating and sending a diff usually requires more computational resources than it does to simply send a page, diff-requests will typically have higher density and will lead to a more potent attack. High-density attacks necessitate less volume than an equally potent low-density attack.

While we are coining a new term (high-density resource-consumption attacks) we are not identifying a new form of attack. High-density DoS vulnerabilities are often found in real-world software and are exploited in the wild. High-density attacks have been described throughout the literature using related terms, which we discuss in Section VI. We discuss existing, complementary techniques for defending against high-density attacks also in Section VI.

Our focus on high-density attacks is motivated by their worrisome features. First, high-density attacks empower otherwise weaker adversaries. With a low-density attack, an adversary may need to use a large amount of their own computational resources to mount a successful attack, motivating techniques such as *distributed denial of service* (DDoS) attacks using botnets. However with a high-density attack the adversary typically doesn’t need a botnet; a successful attack is often possible using a single computer—or perhaps even a single request [1].

High-density attacks are also worrisome because they may produce fewer observables as a result of their lower volume (e.g. less attack traffic produces fewer observables). This makes it more difficult to detect and attribute high-density attacks using automated techniques.

Fortunately for defenders, powerful high-density attacks usually need to exploit specific algorithmic weaknesses in the victim’s system. Because of this, it is not typically possible to develop generic attacks that work well against a variety of victims. This feature also makes high-density attacks more sophisticated and less straightforward to develop than low-density attacks. These limitations (for the adversary) are likely the reason why low-density attacks are more prevalent on the Internet today.

Although low-density attacks represent the modus

operandi today, we speculate that high-density vulnerabilities could become the new “low-hanging fruit” as defenses against low-density attacks become increasingly operational. Furthermore, high-density vulnerabilities—especially *algorithmic-complexity* vulnerabilities [2]—could become more common as the logic of web-applications and service-oriented architectures becomes increasingly complex [3], [4], [5], [6].

Due to the prevalence of low-density attacks on the Internet, most DoS-defense research has focused on low-density attacks. Unfortunately, most of these techniques are insufficient for protecting against high-density attacks since these techniques are often tailored to the high-volume properties of the attack. For example, limiting the rate of the attack is usually ineffective since high-density attacks are usually effective at relatively low rates. Similarly, detection methods that rely on observing high-rate traffic may fail to detect a low-rate / high-density attack. We need specialized techniques to specifically defend against high-density attacks.

In this work we investigate a technique that probabilistically guarantees graceful degradation in the face of zero-day high-density attacks. The mechanism works by monitoring the system for overload conditions, and evicting greedy requests when the system becomes overloaded. This ensures that non-greedy requests will receive their fair share of resources. Since legitimate requests are less likely to be greedy, the server will tend to evict malicious requests more often—creating room for legitimate requests. Previous work has investigated this general idea in non-adversarial settings [7], [8]. To the best of our knowledge though, we are the first to investigate the application of this approach in an adversarial environment.

This paper’s primary contribution is the investigation of a technique that provides servers with provable graceful degradation in the face of *high-density* resource-consumption attacks. Specifically:

- 1) We propose the application of the approach to defend against high-density attacks. Previous work has investigated the application of the approach to busy Internet services (that experience non-malicious overload) and real-time systems [7], [8].
- 2) We formally analyze the application of this technique in an adversarial setting, and develop a formal security guarantee. To illustrate the types of guarantees the technique can provide, consider an example taken from our case study. The guarantee assures that with probability at least 0.99, the server will complete at least 88% of legitimate requests (that have been admitted into the server) within 0.25 seconds (see the example in Section III-N).
- 3) We investigate the practical considerations of applying this technique in an adversarial environment. In particular, the adversarial environment motivates particular

design choices regarding the eviction policy.

- 4) We present a case study of the application of our defensive technique to one of the MediaWiki diff implementations. Experimental results using Wikipedia data confirm our theoretical findings and demonstrate that our technique is practical.
- 5) We analyze the real-world limitations of the defensive technique. In particular, we identify ways in which the technique’s assumptions can be violated in practice and how such violations could affect the mechanism’s effectiveness in practice.

This paper also makes two secondary contributions.

- 1) We define three metrics that measure resource-consumption attacks: mass, volume, and density. These metrics measure the *mechanism* of an attack as opposed to, for instance, the *effect* of an attack. We motivate the significance of these metrics and discuss their relation to other concepts in the literature.
- 2) We provide a survey that relates previously disjoint research on high-density resource-consumption attacks.

The rest of the paper proceeds as follows. In the next section (II) we define the concepts of mass, volume, and density as they relate to resource-consumption attacks. In Section III we describe and analyze the defensive technique. Section IV presents a case study of the application of the technique to Wikipedia. Section V discusses the pragmatics of applying the technique in operational environments. In Section VI we survey related work. In Section VII we outline future work to improve the defensive mechanism. We conclude in Section VIII.

II. RESOURCE-CONSUMPTION METRICS

We define three related metrics that measure important properties of resource-consumption attacks. These metrics measure the *mechanism* of attacks, as opposed to the *effect* (or other aspects) of attacks. In this section we also discuss the significance of these metrics. In the related-work section (VI) we discuss the relation of these metrics to existing concepts in the literature.

Definition. *Mass is an amount of resources consumed by a resource-consumption attack.*

Definition. *Volume is an amount of communication taken by the adversary to conduct a resource-consumption attack.*

Definition. *Density is an amount of resources consumed per unit of communication (essentially mass divided by volume).*

Volume may be measured a number of ways, depending on the specifics of an attack. With a web-based attack, volume could be the number of web requests the adversary sends or it could be the number of bits in the malicious web request(s). For local attacks (as in the case of a locally planted zip bomb [9]), volume could measure the size of the malicious input(s).

The concept of volume, defined here, is backwards compatible with the informal notion of volume as commonly used in the literature. We also note that whether or not an attack has high density is necessarily a relative assessment.

While these definitions are somewhat vague in the general setting, they can be made concrete when applied to specific attacks. For instance, in the formal analysis of our defensive technique we formally define mass, volume, and density.

A. Significance of metrics

We are motivated to define these metrics in order to have a clear notion of attack density. Density is a useful concept because it highlights one of the most salient features of high-density attacks—the fact that an adversary can consume a large mass of resources through a relatively small volume of communication.

Perhaps more significantly, density is an important metric because the efficacy of defensive techniques often depends on the density of attacks: techniques tailored to work well against low-density attacks often do not also handle high-density attacks (as discussed in Section VI). Further, the density of an attack affects its observability as discussed in the introduction.

Lastly, attack volume is typically positively correlated with the economic-cost of an attack; high-volume attacks generally necessitate more adversarial resources. Thus, high-density attacks are often economical for the adversary.

Our choice of physical metaphor helps us to intuitively understand the relationships between the metrics by leveraging our understanding of physical relationships. Other fields of computer science use similar physical metaphors to good effect. For example, a popular metaphor in distributed computing relates surface area to communication and relates volume to computation.

Although we focus on denial-of-service attacks in this work, it is important to note that a high-density attack is not necessarily a *denial-of-service* attack. The term “denial of service” merely refers to the goal of an attack. The term high-density refers to the mechanism of the attack. High-density attacks may be conducted for goals other than denial of service. For example, consider a cloud-computing scenario such as Amazon’s Elastic Beanstalk where computational resources are automatically allocated whenever needed—at a price. A high-density attack could cause the victim to use much more resources than anticipated, increasing their cloud-computing bill maliciously. No users would be denied service in this attack.

III. DEFENSIVE TECHNIQUE

In this section we describe the defensive technique, both formally and informally. We present our theoretical model, describe the defensive technique, and prove two related security guarantees. The section concludes with an example guarantee taken from our case study.

A. Requests

We model a server as a process that takes a series of requests as input $(p_1, p_2, p_3, \dots, p_n)$ and produces a series of associated responses as output $(q_1, q_2, q_3, \dots, q_n)$.

A request p is a record with four fields: t, d, r, m , where $p.t$ is the time the request entered the server (in seconds), $p.d$ is the total amount of CPU-time needed to complete the request (i.e. the *density* of the request, possibly infinite), $p.r$ is a the mount of CPU-time the request has received (initially zero), $p.m$ is a marker indicating whether the request is malicious or legitimate. The server cannot access the malicious marker, $p.m$, and can only know the density, $p.d$, once the request completes.

Since the CPU-received field, $p.r$, may vary with time we also index each request according to time: p_k^t is the k th request, at time t .

B. Inter-arrival assumption

We make the following important assumption about incoming requests: each request is separated by a minimum inter-arrival time I . Formally, for all $a < b$, $p_b.t - p_a.t \geq I$.

The minimum separation time, I , could result from the inherent throughput of system. For example the speed of the server’s NIC may be the limiting factor that determines I . Often though it will be desirable for the server to deliberately control I using an admission-control system.

C. Responses

For each request p_k there is a response q_k , which is a record with a single field l , where $q_k.l$ is the latency of the response (the amount of time that elapsed between p_k entering the system and q_k exiting the system). The latency of a response may be either a non-negative real number (if the request has completed), the value *unknown* (if the request is still processing), or the value *evicted* (if the request was evicted by the server).

Initially the latency of all responses is *unknown*. Whenever the server emits a response q , it updates the latency field, $q.l$, for that response. Since responses vary with time, we also index each response according to time: q_k^t is the k th response, evaluated at time t .

D. Mass, volume, density, and time

In this model *mass* is an amount of CPU-time consumed by requests, *volume* is measured in number of requests, and *density* is a measure of CPU-time per request. Since the CPU resource automatically replenishes itself at a constant rate, it makes sense to measure the potency of attacks with regards to time. For example, CPU-seconds consumed per wall-clock second (mass per second) is more relevant for measuring attack potency than just mass alone. Likewise, requests-per-second (volume per second) is more relevant than just considering the total number of requests. Density, however, is time-independent since mass/second per volume/second is just mass/volume.

E. Adversary model

We assume the adversary is able to control a specific proportion of requests, say 95%. These requests are tagged as malicious and the adversary may choose the density of its requests. In other words, the adversary’s only capability is to choose the value of $p.d$ for malicious requests.

The adversary has no other capabilities. The adversary may not influence the CPU consumption of legitimate requests, may not affect the overhead of the system, and may not affect when requests sleep. Also, we assume the adversary may not prevent legitimate requests from entering the system. We leave it to future work to remove this last assumption by incorporating an admission-control system in the model.

This model represents an adversary who can exploit a high-density vulnerability—a way to cause high CPU consumption within individual requests. The adversary’s best strategy is to conduct a high-density resource-consumption attack by causing high CPU-consumption with individual requests. However, a low-density attack can also be conducted in this model. If the adversary is unable to identify a vulnerability that allows her to craft high-density requests, then the adversary can send a high-volume of average-density requests. If the adversary can send requests at a sufficiently high rate (i.e. if I is sufficiently low) then a low-density attack can still cause a denial of service.

F. Server operation

In our model, time is continuous. The variable τ represents the wall-clock time, which starts at 0 and is incremented as time passes.

The server uses a pool of T threads (the *thread pool*). At any given time each thread is either available or is assigned to a particular request. For simplicity’s sake we assume all threads are initially available, though this assumption is not strictly necessary. When a new request arrives it is assigned to an available thread, if one exists. The defensive mechanism, describe shortly, specifies what happens when a new request arrives and all threads are busy.

The server makes progress on requests by distributing CPU time to requests in the thread pool. Over any window of time between τ_1 and τ_2 the server may distribute at most $c \times (\tau_2 - \tau_1)$ CPU seconds amongst the requests in the thread pool. The value c is a constant and represents the *utilization* of the system, where $0 \leq c \leq 1$. We refer to the value $1 - c$ as the *overhead* of the system. When a request receives CPU-time it’s CPU-received field, $p.r$, is incremented accordingly. If at time τ , $p_k.r$ is incremented such that $p_k.r \geq p_k.d$, then the request completes and the server emits a response q_k with $q_k.l = \tau - p_k.t$.

The server may also choose to evict requests before they complete, in which case the request p_k is removed from the thread pool and $q_k.l$ is set to *evicted*.

G. Scheduler assumption 1

We assume the utilization constant, c , is fixed and known.

H. Scheduler assumption 2

We assume the scheduler is fair: requests will receive their fair share of CPU time. Formally, we assume the scheduler provides the following guarantee: if the server does not evict a request p during the first TI seconds that p is in the system, then p will receive at least cI CPU-seconds during the first TI seconds p is in the system. Many real-world schedulers can be used to satisfy this assumption in practice, as long as the inter-arrival assumption holds and requests don’t spend too much time sleeping.

We refer to the value cI as the *budget* for requests. We refer to the value TI as the *deadline* for requests; when the latency of a response is less than TI the response is said to be *on time*, otherwise it is *over time*.

I. Defensive mechanism

If a new request arrives and all threads are busy, then the server evicts a request in order to make room for the new request. The server may evict any request as long as the request has received its budgeted amount of CPU. Requests that satisfy $p.r \geq cI$ are said to be *eligible for eviction*. In our implementation, we chose the policy of evicting the request with greatest CPU usage, but other policies also make sense. We motivate our decision in Section IV-C. Lemma 1 proves that there is always a request that is eligible for eviction.

J. Request-distribution assumption

We assume that the density of legitimate requests follows a known probability distribution. Specifically, we only need to know the probability ρ that a legitimate request is over budget. Formally, we assume all p_k are independently and identically distributed with $Pr(p_k.d \geq cI) = \rho$.

K. Service metric

Our probabilistic security guarantee says that with high probability, our service metric will meet a minimum threshold. The service metric measures the proportion of legitimate requests that finish on time. Formally, we define our service metric M as follows. Let n' be the number of legitimate responses with latency $\leq TI$, and let n be the number of legitimate requests. $M = n'/n$.

It only makes sense to take this measurement once all the the requests have had a chance to receive their share of the CPU, i.e. when $\tau \leq p_k.t + TI$ (where p_k is the last request).

L. Analysis

Lemma 1. *If (1) the inter-arrival assumption holds and (2) the scheduler assumptions hold, then if a request arrives and all threads are busy, there is at least one request that is eligible for eviction.*

Proof: The only time a request can be evicted is when a new request arrives and all threads are busy (i.e. there are T requests currently in the system). Assume this occurs. Since requests can enter the system at a maximum rate of one per I seconds (by the inter-arrival assumption), then the request that has been in the system the longest has been in the system for at least TI seconds. By scheduler-assumption 2, this request has received its budgeted amount of CPU and is eligible for eviction. ■

Corollary 1. *If (1) the inter-arrival assumption holds and (2) the scheduler assumptions hold, then when a request requires CPU-time less than cI , it will complete within TI seconds.*

The following Weak Security Guarantee (Corollary 2) is presented because it is easy to intuitively understand. The Probabilistic Security Guarantee (Theorem 1) is stronger since it gives a lower bound on the probability that a particular service level will be met.

Corollary 2. Weak security guarantee. *If (1) the inter-arrival assumption holds and (2) the scheduler assumptions hold, then when x proportion of legitimate requests are under budget then the service level M will be at least x .*

Theorem 1. Probabilistic security guarantee. *If (1) the inter-arrival assumption holds, (2) the scheduler assumptions hold, and the (3) request-distribution assumption holds, then $Pr(M \geq m) \geq F(n(1 - m))$ where F is the cumulative distribution function of the binomial distribution with parameters n, ρ (where n is the number of legitimate requests and ρ is the probability that a legitimate request is over budget).*

Proof: If n' is the number of under-budget legitimate requests, then at least n' legitimate requests will finish on time (by Corollary 1). Then at most $N = n - n'$ legitimate requests will fail. Since we assume n is fixed and known, then N is distributed binomially with parameters ρ and n (by the request-distribution assumption). Simple algebra shows that $Pr(N \leq x) = Pr(\frac{n'}{n} \geq 1 - \frac{x}{n})$. Since it is possible that over-budget requests will finish on time, then the proportion of successful legitimate requests will be greater than (or equal to) the proportion of under-budget requests; i.e. $M \geq \frac{n'}{n}$.

Therefore $Pr(M \geq 1 - \frac{x}{n}) \geq Pr(\frac{n'}{n} \geq 1 - \frac{x}{n}) = Pr(N \leq x)$, which is equivalent to $Pr(M \geq m) \geq Pr(N \leq n(1 - m))$. Thus $Pr(M \geq m) \geq F(n(1 - m))$ where F is the cumulative distribution function of the binomial distribution with parameters n, ρ . ■

Corollary 3. *If (1) the inter-arrival assumption holds, (2) the scheduler assumption holds, and the (3) request-distribution assumption holds, then with probability at least β , the service level M will be at least $1 - \frac{F^{-1}(\beta)}{n}$, where*

F^{-1} is the quantile function of the binomial distribution with parameters n, ρ .

M. Scope of guarantee.

Our security guarantees only apply for legitimate requests that are admitted into the server. It is thus important for real-world implementations to ensure that the adversary is not able to prevent too many legitimate requests from entering the server. In future work we plan to consider an admission-control system as a component of our system, which should allow us to provide end-to-end guarantees.

N. Example

The following example is taken directly from our case study. Assume there are 5000 requests and that 95% of requests are malicious; there are thus 250 legitimate requests ($n = 250$). Assume each malicious request requires about 18 seconds of CPU time. Assume there is 85.65% system overhead (i.e. $c = 0.1435$), the minimum inter-arrival time is 0.05 seconds ($I = 0.05$), and there are 5 threads in the thread pool ($T = 5$). Each request then has a budget of $cI = 0.007175$ CPU seconds. Assume each legitimate request requires less than the budgeted amount of CPU with probability 92.4%, i.e. $\rho = 0.076$.

According to the Probabilistic Security Guarantee (specifically, the corollary) there is at least a 99% chance that the service level will be at least 0.884; i.e. there is at least a 99% chance that at least 88.4% of admitted legitimate requests will complete within $TI = 0.25$ seconds.

Notice that the proportion of malicious requests and the density of the attack do not affect the guarantee (as long as $n = 250$). However, these properties of the attack will affect how close the actual service level approaches the theoretical minimum.

IV. CASE STUDY: WIKIPEDIA DIFF

A. Vulnerability of diff

To investigate the practicality of our technique we performed a case study using a *diff* routine as our example server operation. A *diff* routine takes two text files as input, compares them, and outputs a summary of the differences between the files. Web applications, such as Wikipedia, often allow users to compare web pages using a *diff* routine (especially for comparing different versions of the same page). What is interesting about *diff* routines is their algorithmic complexity.

To provide an accurate and precise summary of differences, *diff* algorithms must solve the *longest common substring* (LCS) problem—which is difficult to solve efficiently. There exist myriad solutions to the LCS problem but they are all quadratic one way or another [10].

Because of this inefficiency, *diff* routines are alluring targets for high-density attacks. An adversary can attempt an attack by requesting that the server compare two files

that the adversary knows a priori will effect worse-than-average algorithmic performance. This is the exact attack we use in our case study. High-density attacks that cause resource consumption by effecting worse-than-average algorithmic performance are known as *algorithmic-complexity attacks* [2].

B. The Wikidiff2 routine

Wikipedia uses the web-application software MediaWiki to run its service. MediaWiki can be configured to use any of a variety of diff implementations. The current favored implementation is Wikidiff2, which is implemented in C++ and is based on MediaWiki’s previous PHP diff routine [11]. The main motivation for supplanting the previous diff routine was to have robust performance during worse-than-average comparisons. Specifically, the author of Wikidiff2 noted that the previous implementation was vulnerable to DoS attacks due to its poor performance [12]. However, while Wikidiff2 is much faster than its predecessor, it still uses a quadratic diff algorithm and is thus vulnerable to high-density attacks. The source-code package for Wikidiff2 comes with two 2.3 MB Chinese text files that, when compared, cause orders of magnitude more CPU consumption than typical diff requests. We used these files to conduct attacks in our case study.

C. Experimental setup

Server software. Rather than running a full MediaWiki instance as our server, we implemented a custom server in $\sim 2,400$ lines of C code that only performs the diff operation. This approach gave us greater flexibility in implementing our defensive mechanism and allowed us to isolate the resource-usage of the diff operation. The server runs a thread that generates a new random diff request every I seconds. Legitimate (non malicious) requests compare the two most recent versions of a randomly selected page. Requests are drawn according to a corpus of wiki-page source files.

One important difference between our server and the MediaWiki application is that MediaWiki caches the results of diff operations, while our server does not. To conduct a high-density attack against MediaWiki, it would be necessary for the adversary to continually invalidate the diff cache by updating the wiki-pages it is requesting diffs for. Otherwise, diff requests would result in the efficient look up of a cached diff result (i.e. the malicious requests would have low density).

Environment. We ran experiments on a two-core 2.40 GHz MacBook Pro with 4 GB of memory, running a modified version of Linux 2.6.35 (modification described shortly).

Assumptions. Our experimental setup satisfied most of our mechanism’s assumptions: (1) Diff requests are semantically independent; however, caching effects and similar phenomena may cause operations to perturb each other to a limited extent. (2) The inter-arrival assumption is enforced by the

server. (3) Linux’s Completely Fair Scheduler approximates a generalized-processor-sharing scheduler; as long as the inter-arrival assumption holds the Linux scheduler will approximately satisfy scheduler-assumption 2.

One key assumption was violated (scheduler-assumption 1), which led to violations of our security guarantee in some experiments. While we were able to accurately approximate the system overhead for a thread-pool size of one, limitations with our implementation prevented us from directly approximating c for other thread-pool sizes. Thus we made the assumption that c remained fixed across all thread-pool sizes—which turned out to be fallacious.

Defensive-mechanism design. There are two main design decisions we had to make: (1) deciding how to choose requests for eviction, and (2) deciding what to do with evicted requests.

We discuss the former design decision first. When the server must evict a request, there may be several requests eligible for eviction. We consider two policies: evict the request that has used the most CPU-time, or evict the request that has been in the system the longest. The security guarantees hold either way. The policy is nevertheless important since it affects the overhead of the system and may bias eviction to malicious requests (which in turn affects the ultimate service level).

The benefit of evicting requests based on their time spent in the system is that it is more straightforward to implement since CPU usage is more difficult to monitor (at least on modern Linux kernels). Despite this, we chose to evict requests based on CPU usage because we believe it may provide a better service level, though we haven’t quantified or proved this in our analysis. The intuition behind our assertion is this: it is possible that the oldest request may not be the request that has used the most CPU (if that request spent more time sleeping than other requests). If the adversary attempts to maximize attack density, then requests that have used more CPU should be more likely to be malicious. By choosing to evict the request that has used the most CPU, we believe it will bias evictions towards malicious requests (an analysis of this hypothesis remains for future work).¹

The second design decision we made was choosing what to do with evicted requests. The simplest implementation is to simply remove requests from the system once they are evicted—which is what we chose to implement. Alternatively, we could have chosen to move evicted tasks into a low-priority thread pool. De-prioritized requests only receive access to the CPU once non-evicted requests have received their share of the CPU. Thus, keeping evicted requests in a low-priority thread-pool gives over-budget legitimate requests a chance to finish without adversely affecting the

¹One way this hypothesis is violated would be if the adversary clogs the thread pool with requests that do nothing but sleep. In this case, it would be better to evict requests that had been in the system the longest.

scheduler guarantee (scheduler assumption 1). However, in an adversarial environment the majority of evicted requests will be malicious, causing the low-priority thread pool to become saturated with malicious requests. We expect that any legitimate requests in the low-priority thread pool would receive only a small amount of CPU over a large amount of time. Thus we do not expect a low-priority thread pool—alone—to yield practical improvements.

However, a low-priority thread pool could be useful if evicted requests use a *back-up* diff algorithm that is made more efficient by yielding approximate results instead of exact diffs. Such an approach exemplifies the defensive benefit of algorithmic diversity. Previous work has used algorithmic diversity to defend against algorithmic-complexity attacks [6]. This idea seems to have been first proposed as a way of dealing with non-malicious overload for Internet web sites [13].

Defensive-mechanism implementation. The main challenge in implementing our mechanism was measuring the CPU usage of requests. Linux reports CPU usage for requests through the `proc` file system. This is undesirable in our setting since, during attacks, the server must frequently poll the kernel for CPU usage in order to make eviction decisions. Using the `proc` file system in this manner is inefficient since the server must open, read, and close a file, then parse a string to find the usage for a particular request. Furthermore, the kernel only reports CPU usage at a resolution of a hundredth of a second—despite the fact that the kernel internally tracks CPU usage at millisecond resolution.

To improve the resolution of CPU usage we modified the Linux kernel² so that it reports CPU usage at millisecond resolution through the `proc` filesystem. The overhead imposed by the `proc` filesystem is a significant factor that contributes to overhead in our implementation. In future work it will be important to measure CPU usage more efficiently. We expect that further kernel modifications will make CPU-usage queries as efficient as wall-clock latency queries.

Corpus of wiki-page source files. In order to perform diff operations that are representative of real-world usage, we performed diffs using wiki-page source files taken from Wikipedia. Based on the assumption that the most popularly viewed pages are also the most popularly diffed pages, we choose our pages according to their viewership popularity. Specifically, we analyzed Wikipedia’s publicly available logs for the hour between 3:00pm and 4:00pm on January 2, 2011.³ Page popularity during this hour implies a probability

distribution, which we used to generate 891 requests.⁴ Due to the long tail of the popularity distribution, almost all of the requests were unique; however, seven pages were sufficiently popular that they each occur twice in the corpus.

D. Experimental results

Summary. The experimental results demonstrate several points. First and foremost, our security guarantees held when the system was appropriately parameterized. The defensive technique is able to effectively prevent denials of service from high-density attacks. Second, the experimental results demonstrate the importance of accurately measuring the system’s overhead; when the overhead measurement was inaccurate, the security guarantees were violated. More generally this experience shows the importance of carefully measuring the system’s parameters and not over-generalizing the results of trial runs.

All measurements presented are the results of averaging 10 experiments results. Non-attack experiments contained no malicious requests, while in attack experiments each request has probability 0.95 of being malicious. Experiments were run with 5000 tasks; during attack experiments there were about 250 legitimate requests.

CPU-usage distribution. Figure 1 presents the CPU usage of each diff operation in the corpus. Most legitimate requests require little CPU-time to complete; even the most expensive legitimate request consumes only 0.045 CPU seconds. This stands in stark contrast to our malicious request, which consumes about 18 seconds of CPU time: about 400 times more CPU-time than the most expensive legitimate request in our corpus.

Measurement of overhead constant, c . Table I presents our measurements of the overhead constant. We found that request inter-arrival time (I), the presence of an attack, and the size of the thread pool, T , all affected the overhead of the system. However, due to implementation issues we were only able to measure the overhead constant when the thread-pool size was one ($T = 1$).

These results show that setting the inter-arrival time too low ($I = 0.002$) is impractical: new jobs arrive so fast that the system thrashes and little real work is accomplished.

These results also show that the presence of attacks has a significant impact on the overhead of the system. For example, with an inter-arrival minimum of $I = 0.050$ and an on-going attack, the system only spends about 14% of its time actually making progress on requests. Even though the overhead was higher than expected (during an attack), it is sufficiently low such that the resulting CPU budgets support sufficiently good security guarantees, as described shortly.

⁴We attempted to generate 1000 requests but our scraper experienced errors when parsing several pages. Also, the most popular request was for the Main Page but we excluded it from our corpus because we do not expect its viewership popularity to be representative of the popularity of diff requests.

²Specifically, the `do_task_stat` function in `fs/proc/array.c`

³Downloaded from <http://dammit.lt/wikistats/>

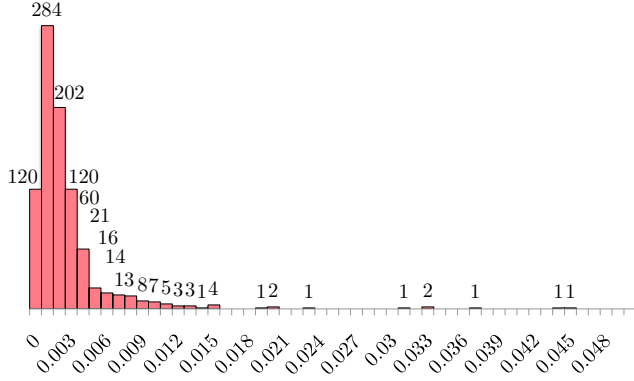


Figure 1. Histogram of CPU usage for the legitimate diff requests in our corpus.

	Non-attack	Attack
$I = 0.002$	0.0003	0.0001
$I = 0.020$	0.8912	0.1230
$I = 0.050$	0.9069	0.1435

Table I
MEASUREMENT OF OVERHEAD CONSTANT, c

	Non-attack	Attack
$I = 0.002$	0.000	0.000
$I = 0.020$	0.989	0.453
$I = 0.050$	0.999	0.924

Table II
PROPORTION OF TASKS UNDER-BUDGET

	Non-attack	Attack
$I = 0.002$	$M \geq 0.000$	$M \geq 0.000$
$I = 0.020$	$M \geq 0.972$	$M \geq 0.380$
$I = 0.050$	$M \geq 0.992$	$M \geq 0.884$

Table III
PROBABILISTIC GUARANTEE: WITH PROBABILITY AT LEAST 0.99,
SERVICE MEASUREMENT WILL BE AT LEAST \bar{M}

	$T = 1$	$T = 5$	$T = 20$	$T = 100$
$I = 0.002$	0.0007	0.9588	0.9980	1.0000
$I = 0.020$	0.9909	1.0000	1.0000	1.0000
$I = 0.050$	1.0000	1.0000	1.0000	1.0000

Table IV
SERVICE MEASUREMENTS, M (NON-ATTACK)

	$T = 1$	$T = 5$	$T = 20$	$T = 100$
$I = 0.002$	0.0000	0.0000	0.0040	0.0190
$I = 0.020$	0.9919	0.9818	<u>0.1004</u>	<u>0.0709</u>
$I = 0.050$	0.9996	0.9996	1.0000	<u>0.1344</u>

Table V
SERVICE MEASUREMENTS (ATTACK), M . RESULTS THAT VIOLATE THE
WEAK SECURITY GUARANTEES ARE UNDERLINED.

	$T = 1$	$T = 5$	$T = 20$	$T = 100$
$I = 0.002$	0.0010	0.9990	1.0000	1.0000
$I = 0.020$	0.9909	1.0000	1.0000	1.0000
$I = 0.050$	1.0000	1.0000	1.0000	1.0000

Table VI
PROPORTION OF LEGITIMATE REQUESTS THAT FINISHED (NON-ATTACK)

	$T = 1$	$T = 5$	$T = 20$	$T = 100$
$I = 0.002$	0.0000	0.0891	0.2150	0.5636
$I = 0.020$	0.9919	0.9947	0.7692	0.6939
$I = 0.050$	0.9996	1.0000	1.0000	0.7700

Table VII
PROPORTION OF LEGITIMATE REQUESTS THAT FINISHED (ATTACK)

	$T = 1$	$T = 5$	$T = 20$	$T = 100$
$I = 0.002$	0.0040	0.0025	0.0025	0.0025
$I = 0.020$	0.0026	0.0026	0.0026	0.0027
$I = 0.050$	0.0026	0.0026	0.0027	0.0028

Table VIII
AVERAGE LEGITIMATE-REQUEST LATENCY (NON-ATTACK)

	$T = 1$	$T = 5$	$T = 20$	$T = 100$
$I = 0.002$	<i>nan</i>	6.3559	13.9759	71.0581
$I = 0.020$	0.0070	0.0294	3.0713	77.4134
$I = 0.050$	0.0073	0.0235	0.0894	77.6232

Table IX
AVERAGE LEGITIMATE-REQUEST LATENCY (ATTACK)

	$T = 1$	$T = 5$	$T = 20$	$T = 100$
$I = 0.002$	0.0074	0.0052	0.0049	0.0049
$I = 0.020$	0.0052	0.0048	0.0049	0.0050
$I = 0.050$	0.0048	0.0049	0.0048	0.0050

Table X
90TH PERCENTILE LEGITIMATE-REQUEST LATENCY (NON-ATTACK)

	$T = 1$	$T = 5$	$T = 20$	$T = 100$
$I = 0.002$	<i>nan</i>	19.4471	27.4413	198.3693
$I = 0.020$	0.0082	0.0364	6.7352	214.6849
$I = 0.050$	0.0081	0.0317	0.1190	212.9397

Table XI
90TH PERCENTILE LEGITIMATE-REQUEST LATENCY (ATTACK)

We believe that the high system overhead associated with attack experiments is primarily due to the inefficiency of polling CPU-usage through the `proc` filesystem.

Proportion of tasks under budget. Using (1) the distribution of CPU-usage for requests and (2) the overhead measurements, we were able to calculate the budget for requests. Recall the budget is the amount of CPU time every request will receive during its first TI seconds in the system and is calculated as cI . Assuming a thread-pool size of one (see Measurement of overhead constant), we found that an inter-arrival time of $I = 0.050$ provided the only good budget during attack; 92% of legitimate tasks require less than cI CPU when $I = 0.050$, $c = 0.1435$, and $T = 1$.

Even though the security guarantees still hold for the other parameterizations, they are less desirable.

Security guarantees. Based on the above measurements we can calculate security guarantees for each of the parameterizations we considered. Table III presents the results of using the Probabilistic Security Guarantee to find the minimum service level with probability at least 0.99.⁵

Experiments. We conducted experiments by varying three parameters: (1) the presence or absence of an attack, (2) the inter-arrival time ($I = 0.002, 0.020, 0.050$), and (3) the thread-pool size ($T = 1, 5, 20, 100$). As discussed, we were only able to measure the system overhead for a thread-pool size of 1. We generalized these measurements and assumed the system overhead would not be affected by thread-pool size.

These settings represent both low-volume and high-volume attacks as well as low-density and high-density attacks. We consider an inter-arrival time of $I = 0.002$ seconds to be a high-volume attack based on the empirical observation that the server experiences an overhead near 100% at this rate. When the thread-pool size is one, even legitimate requests cause a near-complete denial of service. This represents a low-density / high-volume attack—which is alleviated in our experiments by increasing the size of the thread pool.

Since we believed our assumptions were valid, our hypothesis was that the security guarantees would hold during attacks. Our hypothesis held for most experiments, with the exception of some experiments when the thread-pool size equaled 20 and 100. In these cases our assumptions (and security guarantees) were violated. Specifically, we found that the system overhead increased with the thread-pool size, which we did not expect. Table IV (non-attack) and Table V (attack) present our service measurements from these experiments. The instances where our security guarantees were violated are underlined.

Service-level is not the only relevant metric. Recall, the service level measures the proportion of requests that finish “on time.” We also measured the total proportion of

legitimate requests that finished (Tables VI and VII), the average latency for legitimate requests that finished (Tables VIII and VIII), as well as the 90th-percentile for latency for legitimate requests that finished (Tables X and XI).

These results show that when $I = 0.020, 0.050$ most legitimate jobs finished (even though the overhead measurement was wrong for $T = 20$ and $T = 100$). Further, for the four experiments when $I = 0.020, 0.050$ and $T = 1, 5$ (when all our assumptions held) request latency was fast.

V. DISCUSSION

Our formal analysis of the defensive technique proves that it can defend servers from high-density attacks. Our Wikipedia case-study shows that the technique can be effective in practice. In this section we discuss the pragmatics of deploying the mechanism in operational environments. We focus the discussion on the limitations of the defensive mechanism.

Security guarantees provided by the defensive technique only hold when all of its assumptions are valid. Thus the limitations of the technique primarily result from the potential for its assumptions to be violated. We frame our discussion by identifying ways in which assumptions could be violated, and discuss the ramifications of each of these possibilities.

System overhead may not be fixed or known. The case study demonstrates the danger of this assumption. If the overhead of the system is higher than expected than the guarantee will not hold. In the worst case, the system overhead could be so high that the service level drops to zero.

Ideally techniques from real-time systems and formal verification could provide high assurance that the system overhead is known and fixed. However, it is difficult to apply such techniques on contemporary commodity systems. On such systems a careful dependability analysis could provide some assurance that the system overhead is known and fixed [14]. There is also a potential that adversaries may intentionally increase the system overhead. For example, an adversary could conduct a resource-consumption attack against the operating-system kernel—causing the system overhead to increase for the user-space web-application [6]. In our experiments, when the request-rate was too high ($I = 0.002$) this caused the overhead to approach 100%.

Adversary model. It is possible that the adversary may have capabilities beyond what we assumed in our adversary model. We just discussed how it can be possible for attackers to increase the system overhead. There are of course other ways attackers can break our adversary model.

For example, it may be possible for an attacker to increase the density of legitimate requests—causing legitimate requests to be evicted much more often than expected. For example, a vulnerability in the Linux kernel (identified and patched in 2003) allows adversaries to clog the internal

⁵We approximate n , the number of legitimate tasks, to be fixed at 250.

routing hash table with degenerately long linked lists [15]. An attack would cause the kernel to handle certain legitimate traffic flows inefficiently—increasing the density of legitimate flows. In future work we plan to investigate programming-language techniques that can assure that semantically independent requests cannot experience such interference.

We also assume that the adversary cannot prevent legitimate requests from entering the system. In reality, if the adversary increased its attack volume sufficiently, then legitimate requests might not be admitted into the server—and thus would not receive the benefit of the security guarantee. In future work we plan to incorporate an admission-control system into our model which will hopefully allow us to provide end-to-end security guarantees. To illustrate the potential for such an approach we briefly present one admission-control technique that could help. When overloaded, the admission-control filter rejects requests independently and randomly. Clients detect when the server is overloaded through some mechanism and respond by increasing the rate by which they send requests—improving the proportion of legitimate requests and malicious requests that are admitted to the server.

Request-distribution assumption. We just discussed how it may be possible for the adversary to increase the density of legitimate requests. It is also possible that the density of legitimate requests could increase *accidentally*. For example, on June 25, 2009 Michael Jackson’s death caused the volume of legitimate traffic to Jackson’s Wikipedia page to surge. Wikipedia is provisioned to handle such high-volume surges, as they are common occurrences. However, the references section on Jackson’s page was more complex than typical, causing the page-rendering routine to require more computational resources than usual. Because the page was updated so frequently and also due to the lack of a parallelism coordinator, Wikipedia experienced a CPU-overload causing a Wikipedia-wide denial of service [16], [17]. In this situation our defensive technique would still be beneficial, despite the violation of the request-distribution assumption. Our technique would contain the denial-of-service mostly to visitors to Jackson’s page—allowing other Wikipedia users to experience high levels of service.

VI. RELATED WORK

We divide our discussion of related work into two sections: one discussing related terminology and one discussing related defensive approaches.

A. Related terminology

High-density attacks are often mischaracterized or ignored in the literature. A common mistake is to assume that low-density, high-volume attacks are the only form of resource-consumption DoS attack. For example, a 2007 survey on DoS attacks and countermeasures states “[since] the potency

of DoS attacks does not depend on the exploitation of software bugs or protocol vulnerabilities, [potency] only depends on the volume of attack traffic” [18]. Another paper provides a taxonomy of DoS attacks that classifies DoS attacks as either “software exploits” or “flooding attacks” [19]. The paper states that in a flooding attack, “one or more attackers [sends] incessant streams of packets aimed at overwhelming link bandwidth or computing resources at the victim.” Low-volume / high-density attacks do not fit in their taxonomy. Clear notions of mass, volume, and density should help prevent such errors and omissions.

When surveys do accurately characterize high-density attacks, they tend to propose terminology that we believe is less helpful than the terminology we propose. For example, the definition of an *application-based bandwidth attack* according to [18] essentially matches our definition of a high-density resource consumption attack. The term “application-based” is misguided though since (1) a resource-consumption attack against an application does not necessarily have high density and (2) high-density attacks do not necessarily target applications—they may very well target operating systems, middleware, hardware, etc. As another example, a survey of attacks on web services provides a definition for *oversize-payload attacks* that essentially matches our definition of high-density attacks [4]. The term “oversize-payload” is ambiguous though; it suggests that a high-density attack requires voluminous communication.

By defining the metrics of mass, volume, and density we hope to unite the literature to aid our collective understanding of the commonalities and differences that exist among different types of attack.

Though we believe the above cited terms are less helpful, there are several good terms that describe attacks related to (but not equivalent to) high-density attacks. For example, an *algorithmic-complexity attack* is a type of high-density attack that increases resource consumption by effecting worse-than-average algorithmic performance [2]. Chang et al. use the term *input of comma* to describe an extreme high-density resource-consumption attack that can cause a complete denial of service with a single malicious input [1]. Research in DoS attacks against service-oriented architectures has used a variety of terms to describe specialized high-density attacks [4], [20], [21].

B. Related defensive techniques

While little work has been done to specifically address high-density attacks, there exist a variety of techniques that can be used to defend against high-density resource-consumption attacks. This research spans multiple communities, which we survey here. We first describe research that uses the same general approach as ours: evicting over-budget requests during overload conditions. We follow with a discussion of several alternative and complementary techniques.

Evict over-budget requests. Several techniques from the soft-real-time systems community are similar to the one we describe. Most notably, the *constant-bandwidth server* operates using essentially the same mechanism that we describe [7], [22]. Since servers have at least soft-real-time requirements (though often underspecified), we expect that there is much to be gained in future work by applying real-time techniques to systems in adversarial environments. The same approach was independently rediscovered in 2006 by the web-systems community in a paper that investigated evicting greedy web-requests during non-malicious overloads [8]. While the overall approach we investigate in this paper is roughly equivalent to the techniques investigated in the above mentioned research, to the best of our knowledge we are the first to investigate the application of this mechanism in an adversarial setting, from both formal and practical perspectives.

High-volume defenses. The scope of research is too large to survey in this paper, so we only discuss a few key points. First, most techniques either do not work well against high-density attacks or need to be tailored specifically for high-density attacks.

For example, an obvious way to defend against a high-volume attack is to detect and drop malicious requests [23]. There are two issues with this approach: (1) the methods for detecting high-volume attacks need to be adapted to detect high-density attacks since they produce different observables, and (2) while it is usually tolerable to admit a small proportion of low-density malicious requests, a high-density attack may cause a DoS even at low rates. If such techniques are sufficiently adapted, they stand to complement the approach investigated in this paper.

Another well-known approach is to throttle the adversary’s attack volume by forcing all users to solve a computationally expensive “puzzle” for every request they submit to the system [24]. Out of the box, this approach will not work for high-density attacks because this technique only limits the attack-rate to a level that is acceptable for average-density requests. At such a rate, high-density attacks can still be effective. The client-puzzles approach can be adapted to handle high-density attacks by forcing the adversary to solve “variable-density puzzles”—puzzles whose density is proportional to the density of the user’s requests. Once again, if properly adapted this approach would complement our own.

Overload control. Popular web-services are often overloaded on the web, and many techniques have subsequently been developed to help servers deal with overload. We already discussed one such technique that is essentially equivalent to the one we present in this paper [8]. Other techniques may also be effective at defending against high-density attacks; see [25] for a recent survey of such approaches. This area of research is too large in scope to survey here, but we discuss a few interesting systems.

The SEDA framework deals with overload by dividing web-applications into stages and monitoring overload at stage boundaries [26]. When one stage becomes overloaded, the system responds by evicting requests and throttles back its admission control filter. We hypothesize that SEDA has the potential to handle high-density attacks since per-stage admission control should bias evictions towards malicious jobs. It would be interesting to investigate if a SEDA-like approach can be adapted to provide security guarantees similar to the ones presented in this paper.

Another approach from the overload-control area represents a complementary technique. Abdelzaher and Bhatti developed an architecture that responds to overload by using back-up methods that require less resources. For example, if the network link is saturated the system would respond by sending highly compressed GIF images instead of higher-quality JPEG images [13]. In essence this approach turns high-density requests into low-density requests.

Several approaches attempt to bias resource allocation to legitimate requests and throttle the volume of high-density requests. For example DDoS-Shield tries to determine if sessions are malicious based on a model of legitimate workloads [27]. Their “suspicion estimates” influence the scheduler in the hopes of biasing resource-allocation to legitimate requests. In another work, Srivatsa et al. developed a middleware system that uses admission control and congestion control to reduce the impact of high-density attacks [28], [29]. Admission control forces clients to authenticate themselves, allowing the server to track the accesses of each client. The system then use fair queuing to limit the request-rate for admitted clients. Finally, they use congestion control by lowering the priority of clients that consume a disproportionate amount of resources. Both of these approaches may alleviate high-density attacks, but neither provides formal security guarantees.

Static and dynamic analysis A complementary approach to our own is program analysis. For example, Chang et al. used a static information-flow tool to find high-density vulnerabilities in C source code [1].

Pure static approaches, like the one just mentioned, are limited because they cannot leverage information available at run-time. Researchers have thus developed techniques for performing dynamic analyses to address high-density attacks. For example, Cheney and Dahl use static analysis at runtime to estimate the worst-case execution time of SQL queries [30]. Chander et al. use run-time checks to ensure resource operations do not violate resource-quota policies [31]. They perform static analyses to validate that their dynamic checks are sufficient to enforce their policies.

VII. FUTURE WORK

There exists much future work to be done towards developing practical defenses against high-density resource-

consumption attacks. We focus our discussion here on improving the technique presented in this paper.

From a theoretical perspective we plan to target three specific areas for improvement. (1) Perhaps the most important theoretical improvement we foresee would be to incorporate an admission-control system into our model. This would allow us to remove the assumption that the adversary may not prevent legitimate requests from entering the system, and would support end-to-end user guarantees. (2) We also plan to work towards enforcing the assumption that independent requests may not affect each other's computation time using programming-language techniques. (3) Lastly, we observe that the optimal parameterization of the server depends on whether or not the system is overloaded: during normal operation it makes sense to keep the minimum inter-arrival time, I , near zero and the thread-pool size, T , high (say, in the hundreds.) But during attack the most favorable guarantees necessitate a larger value for I and a smaller value for T . We would like to adapt I and T in response to overload, but we must be careful as this may introduce opportunities for adversaries to manipulate these parameters to the defender's detriment.

From a practical perspective, the most important work is reducing the overhead of CPU usage by implementing a better API into the Linux kernel. Also, we plan to conduct more experiments that explore alternative design decisions. Specifically, we plan to (1) run experiments that use an eviction policy on request age instead of CPU usage, (2) test our defensive mechanism on different workload types, particularly IO-bound tasks, and (3) implement a low-priority thread pool with a back-up approximate diff algorithm.

VIII. CONCLUSION

In this work we investigated the application of the "evict greedy requests" strategy to defending against high-density resource consumption attacks. Our research shows that the approach can be effective, provided that the system is appropriately parameterized and the adversary is not able to violate the mechanism's assumptions.

This work is part of a larger research agenda of time-aware and resource-aware computing. Computer science has traditionally abstracted away notions of time and resource usage—which has led to the high-density vulnerabilities we have today [32]. Modern hardware, operating systems, programming languages, and application frameworks do not export resource-usage properties across interfaces and it is increasingly difficult to analyze the resource-usage of systems. High-density resource-consumption vulnerabilities result when software engineers do not understand or anticipate the resource-related behaviors of their systems. When engineers do anticipate such issues it is relatively straightforward to avoid pathological behavior when designing the system.

REFERENCES

- [1] R. Chang, G. Jiang, F. Ivančić, S. Sankaranarayanan, and V. Shmatikov, "Inputs of coma: Static detection of denial-of-service vulnerabilities," in *IEEE Computer Security Foundations Symposium*, 2009.
- [2] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *USENIX Security Symposium*, 2003.
- [3] S. Suriadi, A. Clark, and D. Schmidt, "Validating denial of service vulnerabilities in web services," in *International Conference on Network and System Security*, 2010.
- [4] M. Jensen, N. Gruschka, and R. Herkenhöner, "A survey of attacks on web services: Classification and countermeasures," *Computer Science – Research and Development*, vol. 24, no. 4, pp. 185–197, 2009.
- [5] "Service oriented architecture security vulnerabilities – web services," National Security Agency – Systems and Network Analysis Center Information Assurance Directorate http://www.nsa.gov/ia/_files/factsheets/SOA_security_vulnerabilities_web.pdf.
- [6] M. N. Gagnon, J. Truelove, A. Kapadia, J. Haines, and O. Huang, "Towards net-centric cyber survivability for ballistic missile defense," in *International Symposium on Architecting Critical Systems*, 2010.
- [7] L. Abeni and G. Buttazzo, "QoS guarantee using probabilistic deadlines," in *IEEE Euromicro Conference on Real-Time Systems*, 199.
- [8] J. Zhou and T. Yang, "Selective early request termination for busy internet services," in *World Wide Web Conference*, 2006.
- [9] S. Focus, "Multiple vendor file scanner malicious archive DoS vulnerability," <http://www.securityfocus.com/bid/3027/exploit/>, July 2001.
- [10] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *International Symposium on String Processing Information Retrieval*, 2000.
- [11] T. Starling, "Extension: Wikidiff2 – MediaWiki," <http://www.mediawiki.org/wiki/Extension:Wikidiff2>.
- [12] Starling, "Re: New diff extension," Wikitech-l – Wikimedia developers mailing list. <http://lists.wikimedia.org/pipermail/wikitech-l/2006-February/021600.html>, 2006 February.
- [13] T. F. Abdelzaher and N. Bhatti, "Web content adaptation to improve server overload behavior," in *World Wide Web Conference*, 1999.
- [14] D. Jackson, "A direct path to dependable software," *Communications of the ACM*, vol. 52, no. 4, 2009.
- [15] F. Weimer, "Algorithmic complexity attacks and the linux networking code," <http://www.enyo.de/fw/security/notes/linux-dst-cache-dos.html>, May 2003.
- [16] D. Mituzas, "Embarrassment," Blog post: <http://dom.as/2009/06/26/embarrassment/>, June 2009.

- [17] Brion, "Current events and traffic spikes," Wikimedia Technical Blog: <http://techblog.wikimedia.org/2009/06/current-events/>, June 2009.
- [18] T. Peng, C. Leckie, and K. Ramamohanarao, "Survey of network-based defense mechanisms countering the DoS and DDoS problems," *ACM Computing Surveys*, vol. 39, no. 1, April 2007.
- [19] A. Hussain, J. Heidemann, and C. Papadopoulos, "A framework for classifying denial of service attacks," in *SIGCOMM*, 2003.
- [20] M. Jensen, N. Gruschka, and N. Luttenberger, "The impact of flooding attacks on network-based services," in *The Third International Conference on Availability, Reliability and Security*, 2008.
- [21] S. Padmanabhuni, V. Singh, K. M. S. Kumar, and A. Chatterjee, "Preventing service oriented denial of service (PreSO-DoS): A proposed approach," in *International Conference on Web Services*, 2006.
- [22] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of IEEE Real-Time System Symposium*, 1998.
- [23] R. K. C. Chan, "Defending against flooding-based distributed denial-of-service attacks: A tutorial," *IEEE Communications Magazine*, vol. 40, no. 10, pp. 42–51, October 2002.
- [24] A. Juels and J. Brainard, "Client puzzles: A cryptographic countermeasure against connection depletion attacks," in *Networks and Distributed Security Systems*, 1999.
- [25] J. Guitart, J. Torres, and E. Ayguadé, "A survey on performance management for internet applications," *Concurrency and Computation: Practice & Experience*, vol. 22, pp. 68–106, 2009.
- [26] M. Welsh and D. Culler, "Adaptive overload control for busy internet servers," in *USENIX Symposia on Internet Technologies and Systems*, 2003.
- [27] S. Ranjan, R. Swaminathan, M. Uysal, A. Nucci, and E. Knightly, "DDoS-Shield: DDoS-resilient scheduling to counter application layer attacks," *IEEE/ACM Transactions on Networking*, vol. 17, no. 1, 2009.
- [28] M. Srivatsa, A. Iyengar, J. Yin, and L. Liu, "Mitigating application-level denial of service attacks on web servers: A client-transparent approach," *ACM Transactions on the Web*, vol. 2, no. 3, 2008.
- [29] —, "A middleware system for protecting against application level denial of service attacks," in *ACM/IFIP/USENIX International Middleware Conference*, 2006.
- [30] J. Cheney and M. Dahl, "Resource bound analysis for database queries," in *PLAS*, 2008.
- [31] A. Chander, D. Espinosa, and N. Islam, "Enforcing resource bounds via static verification of dynamic checks," *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 5, 2007.
- [32] E. A. Lee, "Computing needs time," *Communications of the ACM*, vol. 52, no. 5, pp. 70–79, 2009.