Ensuring Content Integrity for Untrusted Peer-to-Peer Content Distribution Networks

Nikolaos Michalakis Robert Soulé Robert Grimm
New York University

Abstract

Many existing peer-to-peer content distribution networks (CDNs) such as Na Kika, CoralCDN, and CoDeeN are deployed on PlanetLab, a relatively trusted environment. But scaling them beyond this trusted boundary requires protecting against content corruption by untrusted replicas. This paper presents *Repeat and Compare*, a system for ensuring content integrity in untrusted peer-to-peer CDNs even when replicas dynamically generate content. *Repeat and Compare* detects misbehaving replicas through attestation records and sampled repeated execution. Attestation records, which are included in responses, cryptographically bind replicas to their code, inputs, and dynamically generated output. Clients then forward a fraction of these records to randomly selected replicas acting as verifiers. Verifiers, in turn, reliably identify misbehaving replicas by locally repeating response generation and comparing their results with the attestation records. We have implemented our system on top of Na Kika. We quantify its detection guarantees through probabilistic analysis and show through simulations that a small sample of forwarded records is sufficient to effectively and promptly cleanse a CDN, even if large fractions of replicas or verifiers are misbehaving.

1 Introduction

Inadvertent and malicious content corruption by misbehaving peers is one of the largest problems in today's peer-to-peer networks. Previous work has addressed the problem in the contexts of file sharing [45, 25] and cooperative storage [10, 12, 28]; this paper addresses the problem for content distribution networks (CDNs). Existing peer-to-peer CDNs, such as Na Kika [20], CoralCDN [16], CoDeeN [46], and CobWeb [43], are currently deployed on PlanetLab, which provides a relatively trusted environment with resources donated largely by the research community. But scaling such systems to the internet at large requires protecting against content corruption by untrusted nodes. Potential attacks range from serving stale content (to save bandwidth), bypassing content transformations (to reduce CPU and memory requirements), to out-right modification, for example, by inserting ads (to generate profit).

The gravity of the problem depends in large part on whether a CDN serves only static content or both static and dynamic content. For CoralCDN, CoDeeN, and CobWeb, which serve only static content, signing the content and response headers at the origin server and verifying these signatures at the clients is sufficient to detect tampering by CDN nodes [4]. Additionally, timestamps can be used to ensure freshness. However, for Na Kika, which also serves dynamic content, hashes, timestamps, and signatures alone cannot establish content integrity—after all, the CDN nodes *are* content producers. Since web-based applications increasingly rely on the dynamic creation and transformation of content [5], CDNs need to support dynamic content and providing a general solution for content integrity becomes a crucial challenge.

This paper presents *Repeat and Compare*, a system for ensuring the integrity of both static and dynamic content in untrusted peer-to-peer CDNs. The key idea behind *Repeat and Compare* is to leverage the peer-to-peer substrate to *repeat* content generation on other nodes and then *compare* the results to detect misbehavior. Principally, this approach is comparable to the use of data replication and comparison through voting or reputation in other peer-to-peer systems, with the crucial difference that *Repeat and Compare* focuses on the replication of computations instead of data. The goal is to enable verifiable and accountable distributed computations similar to the models described in [23, 34, 50, 51].

Repeat and Compare must address three main challenges. First, it needs to observe responses sent to clients, even though clients may lie. Second, it needs to repeat response generation, even though origin servers may supply multiple, conflicting copies of the content. Third, it needs to isolate misbehaving nodes, even though the nodes that "repeat and compare" are not trusted. The underlying source for all three challenges is the same: a "he said, she said" conflict between three mutually untrusting parties: clients, replicas, and origin servers.

To eliminate the "he said, she said" problem, *Repeat and Compare* uses attestation records. Attestation records are included in responses and cryptographically bind origin servers to their content and replicas to their code, inputs, and dynamically generated output. Clients then select a random sample of records and forward them to replicas acting as verifiers. Verifiers, in turn, detect misbehaving replicas by locally repeating response generation and comparing their results with the attestation records. To isolate misbehaving replicas, the system re-

lies on a small core of verifiers, which is trusted by both clients and origin servers, and which distributes a list of suspected replicas. If using a trusted core of verifiers is undesirable, then suspected replicas are distributed using a decentralized trust model. Under this model, each verifier maintains its own list of suspected replicas. Clients learn about misbehaving replicas by contacting verifiers with which they have off-line trust relationships.

We have implemented our system on top of Na Kika and quantify its detection guarantees through probabilistic analysis. We also show through simulations that *Repeat and Compare* effectively and promptly isolates misbehaving replicas, even if a large fraction of replicas and verifiers are misbehaving, and if only a small sample of attestation records reaches well-behaved verifiers.

2 System Design

2.1 Requirements

There are three requirements our system must satisfy. First, users should be able to donate resources to the infrastructure without having to submit their nodes to the administrative control of a central authority. This enables a more flexible replica deployment that can cover a wider range of clients, whether they are geographically dispersed or mobile. Second, the type of dynamic content that replicas can generate should be restricted as little as possible. This enables applications to scale independent of their functionality. Previous solutions for dynamic content generation restrict proxy functionality to a small set of easily verifiable functions that modify original content through add/remove/replace operations [31]. But most of the interesting functionality in web services today involves executing general purpose scripts for content management [3]. Third, a mechanism for ensuring the integrity of content generated within the CDN must not rely on the ends, i.e., clients or servers, because it leads to a tension between scalability and trust. Relying on origin servers to verify the integrity of content delivered through the CDN does not scale. Also, it is easy and cheap to create client or server identities and use them to subvert detection.

2.2 System Overview

To illustrate how *Repeat and Compare* works, we walk through an example. A user browses through her mobile device an online bookstore that aggregates book listings put together by a large number of independent bookstores. To help scale the bookstore application, each bookstore has donated nodes to serve as replicas. Each request from the mobile device may involve transforming media to fit a small screen or sorting a list based on pricing information. If intermediate results have already been generated by other replicas (e.g., media transcoding), they can serve as inputs for generating the final response, thus saving computational overhead and bandwidth.

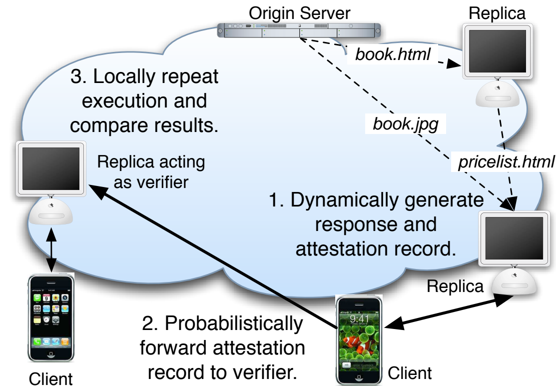


Figure 1: *Repeat and Compare* overview.

Figure 1 shows a high-level view of how *Repeat and Compare* can verify the operation of an untrusted CDN. Clients access content through peers in the CDN that act as *replicas* of an *origin server*. Replicas are used both as caches for static content, such as book items, as well as generators of dynamic content, such as sorted price lists of books [16, 20, 43, 46], while origin servers keep authoritative copies of the content. Replicas locally generate responses if they have all the necessary inputs. Otherwise, they contact other replicas or origin servers for inputs. In addition, replicas can cache dynamically generated content using site-specific scripts [20]. This leads to the flow of content and computations illustrated in the upper right of Figure 1.

At this point, the client has received a response with possibly corrupted or stale content. In the bookstore example, it is possible that a replica sorts search results so that certain books appear at the top to boost the sales for a particular seller. To ensure content integrity *Repeat and Compare* takes an optimistic approach: the client accepts the response, but informs the CDN about it. First, each replica embeds an attestation record in the response. Second, the client forwards the record to a randomly selected replica that acts as a *verifier*. To reduce the overhead imposed on the system, the client forwards only a sample of the records that it sees. Third, to detect misbehaving replicas, the verifier repeats execution and checks if the result matches the record sent by the client.

Verifiers, however, may be malicious. To expose misbehaving verifiers, *Repeat and Compare* uses either a centralized or a decentralized approach. By deploying a small, centralized set of trusted verifiers, *Repeat and Compare* relies on untrusted verifiers to absorb most of the verification load and on the trusted ones to produce the final result. If a trusted set of verifiers is not desirable, then, in order to learn which replicas are misbehaving,

clients can select which verifiers they trust using off-line relationships.

Our design meets the three requirements as follows. Using attestation records to enforce accountability allows users to donate resources to the infrastructure without having to submit their nodes to the administrative control of a central authority. Repeating execution and comparing results enables applications to scale independent of their functionality. Organizing replicas using either a trusted core or decentralized trust relationships helps counter the tension between scaling detection and trusting the outcome of verification.

2.3 Design Space

Comparable to PeerReview [23], our system ensures content integrity through detection rather than prevention (e.g., BFT [7]) for three reasons. First, in contrast to prevention, detection does not impose a bound on the number of misbehaving replicas. As long as there is one well-behaved replica, it can detect every misbehaving replica, since it locally repeats execution. Second, detection is cheaper and can be decoupled from the actual request-response process. This is important since the main reasons for using CDNs are to reduce client latency and bandwidth consumption across the internet. Third, detection is good enough both for bounding the damage over the client population and over time. In addition, it reduces the incentives for misbehavior by exposing misbehaving nodes.

Basically, *Repeat and Compare* treats misbehavior as a failure and thus represents a failure detection system. As a result, it provides *completeness* and *accuracy* guarantees as described in [8]. Completeness ensures that misbehaving nodes will be suspected by the system. Accuracy ensures that well-behaved nodes will not be suspected by the system. To motivate our solution, we present a series of straw man approaches. We show how to overcome the shortcomings of each approach, thus justifying our design. Table 1 summarizes the main idea, benefits and drawbacks for each approach.

Detection by Clients: A natural application of the end-to-end argument [37] would recommend that the client be the verifying end, since it is also the receiving end. This approach is suitable for static content, when the content and response headers are signed by the origin server. But for dynamic content, clients cannot always verify the integrity of received data as it may be expensive to do so. In the bookstore example, a client would need to dynamically generate a price list for different books. This would involve contacting different vendor servers, which nullifies the benefits of the CDN. Another option is for clients to send multiple requests to randomly chosen replicas, similar to sampled voting [28]. However, the only benefit of using this approach is to tolerate a small fraction of misbehaving replicas. If a large fraction of replicas is misbehaving (i.e. more than half), then voting provides neither accuracy nor completeness.

Detection by Spies: An alternative for detecting misbehavior is to have replicas imitate clients and send requests to other replicas to monitor their behavior. Since replicas can misbehave at will, if replica identities are public, direct replica-to-replica monitoring will not work. One way to overcome this problem is to use hidden verifiers. The idea is that a trusted party deploys a set of "spies", nodes whose identities are secret and which imitate regular clients. Detection through spies is attractive because it does not require any modification to the application protocol, including the clients and servers using the CDN.

However, spies are not a feasible long-term solution. Since spies do not know when misbehaving replicas send corrupt responses, they must maintain a constant flow of monitoring requests to detect misbehavior. As a result, it is very likely that, after some arbitrarily long period, every spy that monitors a particular replica will have sent at least one request to that replica. A malicious replica can patiently serve correct responses until a certain timeout, while recording client addresses in a "suspected spies" list. Then it can serve corrupt responses to clients not on the list. Using this strategy, it can remain undetected with high probability. Spies do not have a way to detect that they are suspected because not observing corrupted responses does not imply that a spy was caught. After all, it is also possible that there are no misbehaving replicas in the system. So, either new spies need to be periodically deployed or existing spies need to replenish their network addresses, even if there are no misbehaving replicas. Although this solution ensures accuracy, it is not attractive because in order to provide completeness it leads to an arms race for maintaining spies.

Detection by Informers and Reputation: The alternative is to use responses received by clients. The basic idea is that some clients act as "informers" and volunteer to forward received responses back to a verifier. The verifier detects if a replica is misbehaving by locally generating the response and comparing it with the forwarded response. This idea is attractive because it requires only volunteers to make it work on the existing web. No changes to servers are necessary and, except for the volunteers, clients remain unmodified. However, using informers without protecting the content of responses leads to the "he said, she said" problem. If a verifier sees a corrupt response, it does not know if it came from a replica or was tampered with by the client.

Using a reputation system to compute an aggregate trust score for informers can lessen the "he said, she said" problem by limiting the effects of dishonest informers [25]. A verifier decides what the untampered

Approach	Key Idea	Benefits and Drawbacks
Detection by Clients	Clients verify the integrity and freshness of the data they receive.	Protects static content. Resource-limited clients may not be able to verify integrity of dynamically generated content.
Detection by Spies	A set of spies imitates clients by sending requests and then verifying responses.	No changes to clients or servers, but cannot know if a spy is compromised.
Detection by Informers & Reputation	Verifiers use responses forwarded by clients to compute a replica's trust score.	Relies only on volunteers, but suffers from the "he said, she said" problem.
Detection by Trusted Verifiers using Attestation	Replicas held accountable through attestations. Response verification is performed by a small set of trusted verifiers.	Solves the "he said, she said problem". Does not scale.
Servers as Verifiers	Origin servers verify a sample of the responses using records from informers.	No need for globally trusted verifiers. Overloads popular servers. No ownership for aggregated content.
Detection by Untrusted and Trusted Verifiers	Relies on untrusted verifiers to assist trusted ones in verification.	Scales verification in the common case of correct responses by reducing the probability of detection.
Decentralized Trust	Verifiers keep local lists of misbehaving replicas. Clients learn about replicas using offline trust relationships.	No need for globally trusted verifiers. Detection is slower.

Table 1: Overview of potential approaches for detecting misbehaving replicas, listing the key idea, benefits and drawbacks for each approach.

response is, based on a weighted majority vote from different informers, before it repeats the response generation. In a large-scale environment, however, a reputation system can be gamed by a strong adversary that can gain a large enough foothold inside the volunteer informer group. Large botnets spanning hundreds of thousands to millions of machines make this threat real [47]. As a result, this approach does not provide completeness nor accuracy.

Detection by Trusted Verifiers using Attestation: If responses are signed by replicas, then malicious informers cannot tamper with them, so an informer-based reputation system is not necessary for resolving the "he said, she said" problem. By attesting to the integrity of their responses, replicas are held accountable. In addition, attestations must be linked back to the original data and code, so origin servers must also attest to the integrity of their original data. In the bookstore example, if a book cover image is downsized to fit a handheld, it must be linked back to the original image. So, forwarding attestation records to verifiers is sufficient for ensuring completeness and accuracy.

The next issue is deciding who verifies attestation records. The simplest approach is to use a small set of verifiers that are trusted by both clients and origin servers. However, when most of the records are valid, i.e., they correspond to correct responses, trusted verifiers can be overwhelmed, since the forwarded records grow in proportion to the client population. Because replicas generate content that might depend on client properties or the time of the request, using a cache for consistent records will not prove useful. Even if clients forward only a fraction of received records, verifiers must still handle traffic that is proportional to the number of clients. As a result, trusted verifiers are a bottleneck in the common case and a better verification approach is necessary.

Servers as Verifiers: An alternative is to use origin servers as trusted verifiers for their content. The idea is that clients forward records to origin servers only for responses related to their own content. This is attractive because clients and servers are not required to trust a third party to perform verification. Servers could deny access to their content to replicas that they consider misbehaving. However, this approach has three problems. First, servers need to verify responses proportional to the popularity of the content and, as a result, this approach does not scale well. Second, clients must somehow learn which replicas are misbehaving or not, so that they avoid them. This means maintaining per-server state at the client. Third, in collaborative applications, no single server has ownership of the delivered content. In the example of the collaborative bookstore from Section 2.2, a list of book prices is generated from content that belongs to different vendors. As a result we dismiss this approach.

Detection by Untrusted and Trusted Verifiers: A way to scale verification that does not rely on clients or

servers is to filter valid records by redirecting verification to untrusted verifiers. Untrusted verifiers then forward only invalid records to trusted verifiers (if they forward valid ones they are misbehaving). This solution scales for two reasons. First, since all replicas can perform the same functionality, we can have as many verifiers as replicas in the CDN. Second, a single invalid record is a proof of misbehavior, so the total number of records processed by trusted verifiers is on the order of the number of misbehaving replicas at any given time. Because attestation records are tamper-evident, the worst an adversarial untrusted verifier can do is drop the record of a corrupt response, thus delaying detection. However, if every record seen by a client is forwarded to a verifier, the total traffic to the CDN is essentially doubled. Since using untrusted verifiers already provides only probabilistic completeness, this motivates an approach based on sampled forwarding. Clients only forward an attestation record with probability p.

Decentralized Trust: A globally trusted set of verifiers greatly reduces the communication and computational overhead of detection because results can be publicized and believed by everyone in the system. However, it is not always desirable to have a globally trusted authority, especially when a CDN is built through collaborative efforts and is self-managed [20]. Instead, a decentralized detection mechanism is necessary, where each misbehaving replica is independently detected by each correct replica. Under this model, clients learn about misbehaving replicas by contacting verifiers with whom they have off-line trust relations. To ensure completeness, each misbehaving replica must be detected by all correct verifiers/replicas. As a result, the computational overhead per misbehaving replica is on the order of the number of replicas in the system—after all, each replica will have to verify attestation records by itself.

Decentralized detection may be augmented using gossip initiated either by clients or verifiers. A client can randomly select v verifiers to forward an attestation record, thus increasing the chances that it reaches a "good" one by a v:1 ratio. However, it also imposes a v:1 increase in traffic in the CDN, which is not desirable. Gossip might be more beneficial when used by verifiers. Since verifiers only forward incriminating attestation records to other verifiers, the communication and computational overhead in this case is the same as if each verifier was contacted by a client instead.

Summary: We have explored the design space in order to successively pin down *by whom* and *how* detection is performed. As a result, we deduce three core properties for *Repeat and Compare*. First, solutions that preserve existing servers and clients are not sufficient. Servers must sign their responses and clients must verify them. In addition, replicas must produce attestation records that clients can forward to replicas, which act as verifiers and thus detect misbehavior. Without the accountability enforced by attestation records, there are no detection accuracy guarantees. Second, if clients forward all records they observe, then traffic is essentially doubled. This motivates a sampled forwarding approach, which provides only eventual detection completeness with high probability. Third, there is a trade-off between trust and computational cost. When there are trusted verifiers, the cost for detection is low. When trusted verifiers are not desirable, decentralized detection can offer the same guarantees, but with a cost proportional to the size of the CDN.

2.4 Repeat

Repeat essentially simulates the response generation process illustrated in Figure 1 by recursively generating intermediate results until it outputs the final response. Repeating execution in the presence of non-determinism, external inputs and implicit parameters might not always produce identical results. Removing non-determinism for general purpose computing is impossible. However, web-based architectures are restricted enough to make the task tractable. Client requests are processed independently through well-defined interactions at the HTTP protocol level and applications are relatively stylized, written in scripting languages with bounded functionality.

Identifying external inputs (client request, original content) and configuration parameters (server configuration, library versions) is as important as identifying which code to repeat. Repeating the same code with different inputs, versions of inputs, or configuration parameters will likely produce different outputs. Therefore, external inputs and configuration parameters are uniquely identified by a name, timestamp, and value. They are then cryptographically bound to the code that used them via attestation records. Replicas use this explicit information to set up a runtime equivalent to the one that generated the original response.

Attestation Records

As mentioned earlier, attestation records are a necessary ingredient to preventing the "he said, she said" problem caused by untrusted origin servers and clients. What allows an observer to detect misbehavior using attestation records is a chain of accountability from the client to the server. A broken chain serves as a misbehavior detector, whereas a solid chain proves compliance with the protocol.

Attestation records are bound to either *literals* or *references*. Literals are explicit data values such as system clock times, seeds for random number generators, sensor values etc. They are typically small inputs that come either from origin servers or clients. References

are uniquely named data that need external communication to be fetched. These are data that have been dynamically generated and labeled by replicas as part of the response generation process. Examples include transcoded media and aggregated content. In addition, static content that resides on origin servers is treated as a reference. Although such content could be represented by literals, using references saves space in the attestation record in case the content is large.

In order for accountability not to be compromised, a record must have the following properties:

- It must bind a replica or origin server to the response it generated, so that it cannot later deny it generated that response.
- It must be practically infeasible to generate a record that incriminates another replica.
- A verifier must be able to verify the integrity of the response using only the record and its own execution environment.

To show how attestation records satisfy those properties we explain how they are structured. Literals are of the form

(TYPE, TIMESTAMP, VALUE, PUBKEY, SIGNATURE)

and references are of the form

(TYPE, NAME, TIMESTAMP, VALUE, INPUT-LIST, PUBKEY, SIGNATURE).

In more detail, the different attestation record entries have the following roles.

TYPE. For literals we distinguish types into CONFIG, REQUEST, and ONETIME because each is processed differently. CONFIG records are used to set configuration parameters and initialize the runtime environment. One example is setting pseudo-random number generator parameters such as the algorithm or seed. REQUEST records bind the requester to the generated content and are sent together with the request; they ensure that a client cannot lie about the content it requested. ONETIME records are discussed below. References always have type RESPONSE, since they represent the content returned with a response. The difference between static and dynamic content is determined by checking for an empty input list.

NAME. The name is a unique reference to the content, a URL in our system. Since URLs embed the name of the authoritative server, a replica cannot claim authorship of content it does not own. Names are not necessary for literals, which contain the actual data as part of the value entry.

TIMESTAMP. The timestamp identifies different content versions and is used to ensure the freshness of a response in case content has been updated by the origin server. Its use is explained in more detail in the next section.

VALUE. The value is used to verify integrity; it must match the content produced during verification. For references, the value is a digest of the response. For literals, it is the actual plain-text value. For REQUEST records, it contains client-specific inputs that affect how content is processed (e.g., `User-Agent` for transforming images for cell phones).

INPUT-LIST. An optional list of attestation records declaring the inputs used to generate a record of type RESPONSE. The list contains a record for the executable, configuration parameters, and any external inputs. Using input lists, the RESPONSE record is sufficient to guide the verifier through execution. Notably, the structure of the record resembles the recursive process of content generation in a CDN and thus provides the verifier with all the information necessary for generating the final response from scratch. Note that, besides a constant number of input list entries to identify a replica's execution environment and content processing script, the length of an input list is proportional to the number of aggregated HTTP resources.

PUBKEY. The public key is bound to the identity of the principal that this record speaks for. Any of the existing distributed infrastructures [14, 54] can be used for managing public keys. The certification authority plays no role in the CDN itself.

SIGNATURE. A signature using the principal's private key. It signs all other entries of the attestation record. The signature binds an origin server to its content. It also binds a replica to the original request, the replica's execution environment, any inputs, and the final response. Assuming that an adversary cannot break cryptographic primitives, one must steal a replica's private key in order to incriminate it.

Freshness

To ensure freshness, the client must receive the *latest* version of the content as defined by the origin server. Since both servers and replicas can lie, to determine the latest version, servers must promise not to modify content for a predetermined time period by setting the TIMESTAMP of the RESPONSE record to an expiration time in the future. This ensures that there is only one latest version, the one with a timestamp that has not expired. When a request is made, the client sets the current time as the TIMESTAMP of the REQUEST record. To determine the current time, we assume that all nodes (clients, replicas, origin servers) have loosely synchronized clocks, i.e., all clocks are within skew σ and rates do not diverge. To ensure that clocks remain loosely synchronized, our system trusts an external time service such as NTP. To detect that a server sent multiple conflicting versions, it is

sufficient for a verifier to see two records signed by the server with versions greater than the timestamp of the request (within σ).

Since there is no third party that can observe when a client sent a request to a replica or when a replica sent a request to an origin server, ensuring freshness requires addressing a time-dependent instance of the "he said, she said" problem. If a client forwards a record at time t that contains a timestamp e, where $e < t$, the verifier cannot distinguish whether the replica sent stale content or the client waited until time $t > e$ before forwarding the record. To address this issue, each requester and responder agree on the time of the request through the use of the REQUEST record. The responder accepts the request only if the embedded time is equal to its own time within σ. Since the request record is part of the response record's input list, it is signed by the replica, so the client cannot modify it. Upon receiving the response, the client checks that an attestation record's request time was not modified by a malicious replica.

With this information, a verifier can detect if content is fresh by using the attestation record to check that (a) the timestamp of the request is less than the timestamp of the response plus σ, and that (b) the timestamp of the record is less than the timestamp of each input plus σ. In addition, the verifier needs to perform a sanity check to avoid errors caused by clock skew. It needs to verify that the difference between its current time and the timestamp of each input is greater than σ before it verifies freshness. This check ensures that every node has either transitioned to the newer version of the content or still sees the current one.

Non-Repeatable Data

When repeating dynamic content generation, care must be taken to identify which data intrinsically change over time and, as a result, are not repeatable. We identify two types of such data: one-time values and randomly generated data. One-time values depend on the exact time that they were produced. Examples include sensor measurements and stock prices. The problem with repeated execution is that a verifier repeats the response generation process at a later time, so if it makes a request to a content producer for a one-time input it will get a different value. We avoid this issue through the use of ONETIME literals. We restrict their generation to origin servers because the integrity of these records cannot be verified otherwise.

Similarly, when data are randomly generated, repeating the process will result in different outputs with high probability. To avoid this issue, applications are constrained to use pseudo-random number generators and produce a CONFIG literal with the seed inserted in the VALUE entry. To defend against attacks where a replica "fixes" seeds to reduce randomness, seeds can be provided either by origin servers or clients.

Runtime

Before a verifier repeats execution, it is important to set the correct runtime environment. This involves loading the correct code and server configuration. We use CONFIG records for that purpose. These records are bound to the version of the executable, dynamically loaded libraries, and server configuration files. For the executable and the libraries, the records are bound to a digest of the binary. For configuration files, they are bound to specific parameters that must be the same across all replicas. Scripted code executed by a replica, such as Javascript in the case of Na Kika [20], is supplied as part of the RESPONSE record's input list. If any of the libraries or configuration parameters do not match the records, then the verifier treats a replica as misbehaving.

2.5 Compare

Compare consists of two stages: forwarding attestation records to verifiers and then detecting misbehaving replicas. As explained in Section 2.3, attestation records are forwarded to verifiers via clients. A client first checks that each response contains a consistent attestation record; otherwise, the client drops the response. Then it forwards the record with probability $p < 1$ to a randomly selected verifier. Sending a fraction rather than all the records helps reduce the traffic overhead in the CDN. Because current web clients do not support the notion of attestation records clients either have to be modified or install a local web proxy [33]. Unless a large enough fraction of the client population supports attestation records, replicas have no incentive to produce attestation records. To detect replicas, we adopt both centralized and decentralized approaches.

Centralized Detection

As explained in Section 2.3, if there is an authority that is trusted by both clients and origin servers, then this authority can deploy a small set of trusted verifiers to detect misbehaving replicas. To scale detection, we leverage the massive replication offered by the CDN. Untrusted verifiers receive records from clients, perform *Repeat* and if they detect a misbehaving replica, they forward the incriminating record to a trusted verifier. To prevent malicious verifiers from overloading trusted ones, if the forwarded record is not incriminating, then the untrusted verifier is considered misbehaving and punished (same as any misbehaving replica). As a result, misbehaving verifiers have no incentives to forward "good" records and the traffic seen by trusted verifiers is proportional to the number of misbehaving replicas. By definition, trusted verifiers are trusted by all nodes in the system. Consequently, once a replica has been detected as misbehav-

ing, no further verification of that replica's responses by other replicas is necessary.

Decentralized Detection

If a globally trusted authority is not desirable, then detection can be performed using a decentralized transitive trust model similar to Credence [45]. In Credence, each peer keeps a local voting table for files it has received. It uses the table as a guide to find like-minded peers through statistical correlation of votes. Our model differs from Credence in two ways. First, votes are not cast by manually examining content as in file sharing, but through *Repeat*. Second, the votes are not about content, but about replica status, i.e., misbehaving versus well-behaved.

So, decentralized detection works as follows. Like in the centralized model, replicas act as verifiers. Clients forward attestation records to them and they locally repeat execution to detect misbehavior. If they detect misbehavior, however, they do not forward the attestation record to trusted verifiers as before. Instead, they maintain a local voting table that is initialized with positive votes for every replica. If they receive an attestation record that incriminates a replica then they permanently change their vote for that node to negative. Clients access the CDN by contacting a *friend*, a verifier that they trust through off-line trust relationships. Friends then forward to the clients lists of replicas that they believe are trustworthy, i.e., rank on the top of their lists. Clients use the lists to find nearby replicas to access content. They refresh the list of replicas by periodically contacting their friends.

Punishment and Redemption

Once replicas are detected they are punished. A strict punishment policy would require that replicas be banished. However, this might be too harsh in practice as misbehavior may be the result of human error. We acknowledge that there is no perfect solution to this problem. But, since our system decouples detection from punishment, a real deployment can use application-specific policies to let nodes reenter. For example, in the bookstore application of Section 2.2, one can use tit-for-tat punishment: If a replica donated by a particular vendor misbehaves, replicas block access to the vendor's content (i.e., remove the vendor's books from the price list) for a few hours to decrease sales.

Detection Guarantees

As explained in Section 2.3, failure detectors (as defined in [8]) are characterized with respect to *accuracy* and *completeness*. In terms of accuracy, *Repeat* ensures that no well-behaved replicas are ever suspected, no matter how many records are sent to verifiers. Completeness, however, depends on *Compare* and is only probabilistic. For every corrupt response sent by a misbehaving replica, there is always a positive probability that the replica will be detected.

Eventual completeness is guaranteed only if the system maintains randomness. If clients keep forwarding attestation records to the same verifiers, some misbehaving replicas may never be detected. This is a real possibility for CDNs because clients usually get redirected to nearby replicas. Other than locality, however, completeness is independent of the underlying peer-to-peer topology; as a result, *Repeat and Compare* works with arbitrary peer-to-peer CDN architectures. In practice, it is important to quantify how fast the system detects misbehaving replicas and how much damage they can realize before detection. We quantify *Repeat and Compare*'s eventual completeness guarantees in Section 4.1.

3 Implementation

Our prototype implementation of *Repeat and Compare* builds on four open source packages: the Apache 2.2 web server, the Apache-based Na Kika CDN [20], the Privoxy personal proxy [33], and the OpenSSL cryptographic library [30]. Server- and replica-side functionality is provided by two Apache filters. One filter signs content and comprises 1,000 lines of C code, while the other filter verifies content and comprises 700 lines. Client-side functionality is provided by a 600 line extension to Privoxy, which decouples our implementation from the actual browser used to access the CDN. All three components leverage an attestation record library, which builds on OpenSSL and comprises 4,400 lines of C++ code.

For Na Kika, we modified the *fetchURL* function to add remotely accessed inputs to an attestation record's input list; we also added hooks to get and set the seed of JavaScript's random number generator. Our attestation record library exports a C/C++ API for creating, parsing and validating attestation records using RSA signatures. Content digests are computed using SHA-1. Attestation records are passed between nodes as `X-Attestation` response headers. The configuration and the runtime information is expected to change slowly. So, to amortize the cost per response, it is periodically updated using hashes of the Apache executable, libraries, and configuration files. Replicas simply insert the hashes as inputs in each attestation record. Finally, for the signer module we implemented an in-memory cache to store records for content that have already been hashed and signed.

4 Evaluation

The hypothesis of this work is that *Repeat and Compare* makes fast forward progress at detecting misbehaving replicas even if a very large fraction of replicas are misbehaving. Detection depends on the forwarding proba-

bility p, the number of misbehaving and well-behaved replicas in the system, and the number of corrupt responses a misbehaving replica has sent. To quantify expected system behavior, we analyze how each parameter affects the probability and rate of detection, the time it takes to cleanse the system, and the damage incurred until cleansing is complete.

In practice, network and processing delays also affect the time to detection. To account for these parameters, we simulate a large-scale deployment of both the centralized and decentralized versions of *Repeat and Compare*. Additionally, locality can have a negative impact on our system's effectiveness, as it reduces randomness. In particular, CDNs usually redirect clients to nearby replicas in order to minimize client-perceived latency—but, as a result, they also limit the set of replicas and verifiers visible to each client. We account for the effects of locality through additional simulations.

Since *Repeat and Compare* requires signing all messages and repeating execution, detection can be expensive and affect a CDN's scalability. As a final assessment, we perform a set of micro-benchmarks to experimentally determine how throughput changes when servers generate attestation records and when replicas verify them.

4.1 Detection

Analysis

The goal of our analysis is to abstract away application-specific properties of CDNs and show the core guarantees of *Repeat and Compare* given perfect randomness, and no network and processing delays. We examine a static view of the system, where no new nodes enter or leave (other than those detected). We assume that clients forward records with probability p and that there are f misbehaving and g well-behaved replicas in the system. g is constant over time because our system ensures accuracy. Since a client will receive corrupt content with probability $f/(f+g)$, content integrity is ensured when all misbehaving replicas have been detected. So, we are interested in four properties of our detection mechanism: (a) the probability of detecting a misbehaving replica, (b) the rate of progress made by detection, (c) the time it takes to detect all misbehaving replicas and (d) the damage done (number of corrupted responses) until all the misbehaving replicas are detected.

The probability P_D that a misbehaving replica is detected depends on (1) the client forwarding the attestation record and (2) the client forwarding it to a good verifier. Given a sample of attestation records sent by a replica, P_D depends only on b, the number of incriminating (bad) records sent (good records have no effect). To get a lower bound on P_D, we fix f during the time that b records are sent. For every record sent, the probability that it reaches a good verifier is:

$$P_V = \frac{pg}{f+g}, \text{ therefore } P_D = 1 - (1 - \frac{pg}{f+g})^b$$

If f is small, then P_D improves as p increases, otherwise a large f masks the effects of p. However, even if $p = 1$, P_D can reach 1 only by increasing b.

To analyze the forward progress in detecting misbehaving replicas, we examine how the rate of detection changes dynamically as the misbehaving replicas are detected and removed. To simplify the analysis, we assume that at each time unit each replica serves one client response and that clients contact verifiers randomly. Since inactivity does not affect detection nor corruption and replicas typically have similar capacity, we believe this is a reasonable assumption.

The rate of detection r depends on the number of bad requests sent per unit time, which is equal to f:

$$r = \frac{df}{dt} = -f\frac{pg}{f+g} \quad (1)$$

Since detected replicas are removed, the rate is negative with respect to f. To give an insight of how this affects f over time, we examine how r changes as f decreases. We identify two interesting cases: (a) when f is a large multiple of g, and (b) when f is a fraction of g. As long as f is a large multiple of g, then $r \simeq pg$. Since both p and g are constant, detection makes steady progress. As f decreases and approaches g, the detection rate decreases proportionally to how f decreases. But, even in the worst case, where $f = 1$, the rate approaches p.

To estimate the time T it takes to detect all existing misbehaving replicas, we compute the time it takes to detect all of them except the last by solving the differential equation 1 and then compute the average time to detect the last one using P_V. The reason for not using Equation 1 to compute the total time is that when $f < 1$, t asymptotically reaches infinity as f goes to 0, so it is of no practical use. Our estimate is a lower bound because we do not take into account network or computational latency. Misbehaving replicas are detected and removed within the same time unit they served a corrupt response. We also assume there are F misbehaving replicas initially and during detection no new replicas enter the system. Solving Equation 1 gives:

$$t(f) = \frac{F-f}{pg} + \frac{1}{p}\ln(\frac{F}{f}) \quad (2)$$

So, the total time to detect all misbehaving replicas except the last is:

$$\frac{F - 1 + g\ln F}{pg}$$

To detect the last one it takes on average an additional time:

$$\frac{1}{P_V} = \frac{g+1}{pg}$$

This is because the probability of detection follows a geometric distribution with probability P_V. So, the total time of detection is:

$$T = \frac{F + g + g \ln F}{pg}$$

For large F, $T \simeq F/(pg)$, so the time it takes to remove F misbehaving replicas depends on how large is F compared to the constant pg. For small F, $T \sim 1/p$ so detection is faster for higher values of p.

Given f misbehaving replicas in the system, the damage done by a replica until it is detected (given that it is the first one detected among f) follows a geometric distribution and is:

$$B_1 = \frac{1}{p(\frac{g}{f+g})}$$

To compute, however, the total damage B in terms of corrupt responses sent during time T, we must integrate Equation 2 from 1 to F, because f depends on t. Then, we add the total number of corrupt responses sent by the last misbehaving replica, which is T:

$$B = \frac{F^2/2 + gF + 1/2}{pg}$$

If F is large, then $B \simeq F^2/(2pg) + F/p$. So, the damage incurred is basically affected by F. If F is small, then $B \sim 1/p$. So, the damage is reduced as p increases.

The analysis shows that, if the number of misbehaving replicas is a fraction of the well-behaved replicas, increasing p helps increase the probability and the rate of detection and bound the damage. Setting a high p, however, also increases the traffic overhead by a fraction equal to p, as explained in Section 2.3. In particular, if p approaches 1, then traffic essentially doubles. But as the analysis shows, when f is large, the detection rate is dominated by the number of malicious replicas in the system. In that case, detection has enough forward progress momentum to allow for smaller values of p.

Simulations

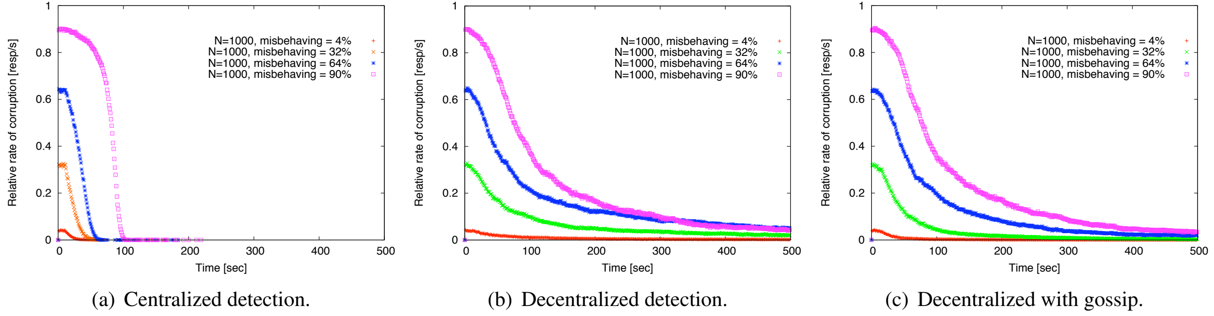
To take into account network characteristics and processing delays, we simulate *Repeat and Compare* using Narses [19] with a topology based on the Meridian all-pairs wide-area latency matrix [42]. We selected 1,000 random nodes as replicas and another 1,000 random nodes as clients. Simulation results are averaged over 10 runs. The request load per replica was set to 1.2 requests per second based on informally gathered statistics from CoralCDN's deployment [15]. Finally, the simulation abstracts away the internals of repeated execution and uses a 5 second computational delay for verifying content. It also limits each verifier to 10 concurrent requests. Both the computational delay and throughput limit are pessimistic when compared to our microbenchmarks in Section 4.3.

Figure 2 shows how long it takes to detect misbehaving replicas for various fractions of misbehaving replicas with $p = 0.1$ both in the centralized and decentralized models described in Section 2.5. Using a small set of trusted verifiers (size=4, Figure 2(a)), misbehaving replicas are detected much faster than in the decentralized detection model (Figure 2(b)). Using gossip does not seem to expedite detection (Figure 2(c)). In our simulation, each verifier forwards incriminating records to 20 randomly chosen verifiers and each forwarded record has a TTL of 2. The reason for gossip's ineffectiveness is that a verifier is more likely to get an attestation record directly through a client than through gossip due to the computational delay of verification. Overall, all three schemes take less than 500 seconds to isolate all misbehaving replicas even if they are 90% of the replica population.

4.2 Locality

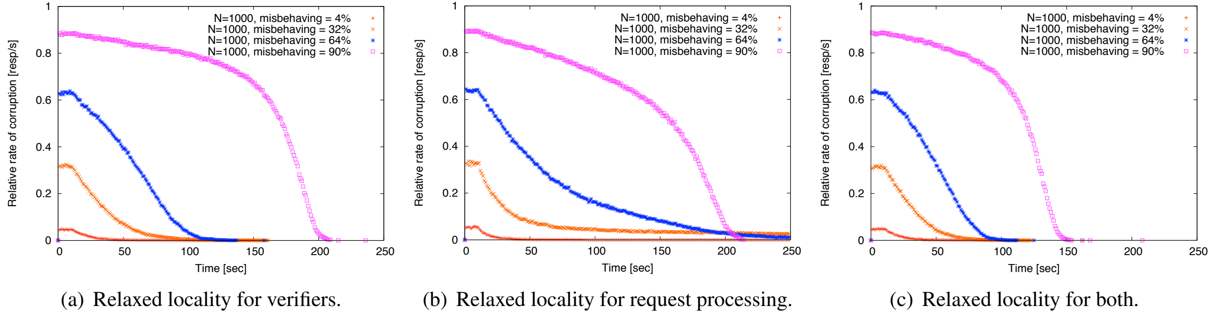
To examine the impact of locality, we use the same simulation environment as before. Since the Meridian matrix contains network distances for each pair of nodes, we use this redundant information to simulate an application-level anycast service by organizing replicas in concentric rings centered at each client starting at 2 ms and doubling the radius at every step. This is similar to how Meridian nodes organize their neighbors [48].

As explained before, there is a tension between locality and randomness. If the application-level anycast service is perfect and always returns the closest replica to the client, then no misbehaving replica is ever detected. This is because the same replica that serves content is also selected as a verifier. So, any incriminating attestation records will always be verified by the very same replicas that generated them. If, however, locality is relaxed by selecting a random node within a 2 ms radius from the client, then detection commences similarly to the random case, as shown in Figure 3. Relaxing perfect locality either when choosing replicas or verifiers is sufficient to ensure progress. There is a subtle difference between the two alternatives. When replicas are chosen randomly but clients remain fixed to the same verifier, the effect is the same as having each verifier scanning its proximity for misbehaving replicas (Figure 3(a)). When verifiers are chosen randomly, but clients remain fixed to the closest replica, then if a misbehaving replica is not the closest node to any client, it will never be detected. Since it is not the closest to any client, it will never serve any requests either, so it cannot do any harm (Figure 3(b)). Finally, a hybrid approach, where both replicas and verifiers are chosen randomly within the proximity of the



(a) Centralized detection. (b) Decentralized detection. (c) Decentralized with gossip.

Figure 2: Comparison of centralized detection, decentralized detection, and decentralized detection with gossip.



(a) Relaxed locality for verifiers. (b) Relaxed locality for request processing. (c) Relaxed locality for both.

Figure 3: Locality vs. randomness. By default, clients are redirected to the closest node; for relaxed locality, a random node within a 2 ms radius from the client is selected.

client detects misbehaving replicas faster (Figure 3(c)).

4.3 Overhead

Using our prototype implementation, we evaluate the overhead of producing attestation records at origin servers and verifying them at Na Kika replicas.

Content Producer Overhead

To characterize the overhead of producing attestation records, we compare a server's throughput and 90th percentile latency when it reaches maximum capacity with and without running the signer module that produces the `X-Attestation` header. The server machine is a 3.2 GHz dual core Pentium D with 4 GB RAM running Fedora Core 6 Linux with a 2.6.18 kernel. Load is generated using the *httperf* [29] web load generator by fetching a single static 2,097 byte document representing Google's home page (without inline images). Since static resources are already well-served by existing web servers, this benchmark illustrates the worst case overhead scenario to content producers that adopt *Repeat and Compare*. Without the signer module the server reaches 1,070 responses per second (rps) with 1 ms 90th percentile latency. With the signer module, the server reaches 443 rps with 2.8 seconds 90th percentile latency. Generating attestation records adds significant overhead to the content producer because it signs each attestation record with its private RSA key. When the signer module caches records for content already signed, the server reaches 1,070 rps with 1 ms 90th percentile latency, i.e. no perceived overhead.

Verifier Overhead

To characterize the overhead of verifying attestation records, we compare the throughput of a Na Kika node running at full capacity with and without running a verifier module. The node dynamically scales an image to fit a 176 x 208 pixel cell phone screen. With the verifier module, the node also repeats execution, generating the same reduced-size image and comparing its digest with the attestation record's digest. The server machine is the same as in the previous benchmark. Na Kika and the verifier run on a 2.8 GHz Intel Pentium 4 PC with 1GB RAM running Fedora Core Linux with a 2.6.9 kernel. Load is generated as in the previous benchmark, but in addition an equal load is generated for verification. Since verification involves checking the consistency of the attestation record using the signer's public key, serving a CPU-consuming page provides a pessimistic estimate of the verification overhead. As a base, running an executable that transforms the image in a tight loop achieves 13.1 transformations per second. Without signing or verification the Na Kika node can serve 6.8 rps. We attribute the extra overhead to Na Kika's Javascript-based

pipeline, which is also a large CPU consumer. When signing is enabled, the node reaches 5.9 rps. When signing and verifying simultaneously, the throughput drops to 2.8 rps. This is expected, since the verifier duplicates the response generation process, thus taking up an equal amount of resources.

5 Related Work

The typical solution for preventing message tampering between clients and servers is end-to-end connection-based encryption using, for example, the SSL protocol. This negates the functionality of the CDN, though, since replicas cannot act on the message. Merkle tree authentication [4] is a proposed alternative, where static cacheable content is signed at the server and verified by the client. Also, digital rights management schemes enable consumers to download content from untrusted distributors [1]. For dynamic content, [9, 31] propose the use of XML-based rules for adding/removing/replacing content, which is limited enough to be easily verified by a client.

Attestation has been used in the context of trusted computing to prove to a third party that a node executes trusted code [32]. The idea is to generate proofs of trust using an attestation chain that cryptographically binds an executable, the data it manipulates, and the underlying operating system to a trusted processor. However, current approaches are limited to attesting to the integrity of the executable's identity, not its functionality. As a result, they cannot detect exploits or misconfigurations (induced by human factor) [18, 39]. Ongoing work [41] focuses on providing more expressive attestations using trusted reference monitors. Since attestations incorporate the operating system, the trusted code depends on a particular OS version, making it harder to accommodate security updates as well as bug fixes and creating a tension between complete attestation and timely upgrades. Our system avoids this tension by focusing only on observable differences. Comparable to our system, the Common Language Infrastructure's *strong names* are used to uniquely identify assemblies, distinguish versions, and provide integrity [13]. Strong names consist of a text name, version number, culture information, a public key, and a digital signature similar to our attestation records.

Minimizing trust placed on nodes in peer-to-peer networks through decentralized reputation systems has been used for file sharing by computing either local values [45] or global values [25], both based on user ratings. As mentioned in Section 2.3, a reputation system can be used for CDNs in order to replace trusted verifiers. However, it is useful only when combined with attestation records and applied to replicas, not clients. Also, sampled voting for content integrity has been used in the context of the peer-to-peer data preservation system LOCKSS [28]. LOCKSS peers verify data integrity by collecting votes from a sample of the population and comparing them with their local copy. Sampled voting combined with repeated execution could provide clients with information for detecting misbehaving replicas without the need for verifiers. However, using verifiers is more suitable for CDNs because unilateral policies for punishing misbehavior are easier to build even if the majority of replicas are misbehaving.

The effects of misbehaving peers can be nullified using byzantine fault tolerance as described in PBFT [7] if the number of misbehaving peers is less than 1/3 of the total number of nodes. However, in a collaborative peer-to-peer CDN, there could be an unlimited number of rational users that donate nodes to benefit from the use of the CDN, but deviate from the service to gain a free ride. This behavior has been modeled as Byzantine-Altruistic-Rational (BAR) [2]. Using the layered approach proposed in [2] for building BAR tolerant systems, one could provide stronger integrity guarantees than our system because *Repeat and Compare* cannot produce proofs of misbehavior when replicas deny service to clients. At the same time, by detecting misbehavior after the fact instead of preventing it altogether, *Repeat and Compare* requires only one replica to generate a response and at most two well-behaved replicas to verify the response (in our centralized model). As a result, *Repeat and Compare* can maintain lower latency and higher throughput than PBFT and BAR systems.

Using repeated execution to detect misbehavior has been used in the context of bug discovery in Rx [35] and worm containment in Vigilante [11]. Our approach differs in that execution is repeated by a separate party, the verifier, rather than the client. Repeated execution in Pioneer [38] is closer to our approach because a trusted platform, the dispatcher, verifies the integrity of code running on an untrusted platform. But since generating a proof of correctness in Pioneer is extremely time-sensitive, it is not suitable for large scale systems.

Our system can benefit from an application-level anycast service that can serve as a controlled entry point for peers that wish to join the CDN. We believe it would be easy to modify systems such as OASIS [17] or Meridian [48] to also redirect clients to verifiers. *Repeat and Compare* has already been integrated with Na Kika [20], which uses a structured overlay to coordinate caches. Structured overlay networks provide robust and scalable coordination strategies [21, 22, 27, 36, 44, 52] and have been successfully used for static content distribution in CoralCDN [16] and Squirrel [24]. However, our system is independent of the coordination mechanism and can also be used by systems such as CoDeeN [46], ColTrES [6], Tuxedo [40] and DotSlash [53] that use domain-specific topologies and algorithms to balance load and

absorb load spikes.

The problem of verifiable, accountable distributed computations has been explored in the contexts of secure information aggregation in sensor networks [34], byzantine fault detection [23], networked services [50, 49], and commercial peer-to-peer computing [51]. Comparable to our approach, these efforts hold both clients and servers accountable for their statements, using some form of attestation record. In departure from our approach, several efforts employ deterministic detection strategies, providing stronger guarantees but requiring more resources. One significant difference (and contribution) of our work is the systematic exploration of the design space.

6 Discussion and Future Work

Comparing the role of determinism in fault detection and fault prevention is instructive. Fault prevention requires that failures are independent and thus introduces non-determinism through n-version programming and dissimilar hardware components. In contrast, fault detection requires that all observable replica behavior be deterministic and repeatable. This limits detection systems, such as ours, to services that do not depend on true randomness. Our implementation meets this requirement by ensuring identical executable versions and runtime environments, including the use of pseudo-random number generators and identical seeds. It is an open issue whether this approach generalizes beyond web applications. We believe that virtual machines may provide the basis for a more general solution.

Our system depends on all requests receiving responses. As a result, replicas cannot be held accountable for refusing service and *Repeat and Compare* cannot prove a denial of service attack. We believe, however, that misbehaving nodes are motivated to serve rather than drop requests in order to maximize damage. As a result, denial of service is not a significant problem in CDNs. Another issue not addressed in this work is the scenario of an attacker strategically changing a small number of high-value responses to maximize damage. This issue is real because our probabilistic detection favors popular content. One possible solution is for origin servers to set a priority value in the attestation records of high-value content, with client forwarding such records with higher probability (e.g., $p = 1$).

Finally, our system does not provide support for databases. To generate attestation records for database reads, records would have to bind results to the corresponding queries and database tables. Supporting database updates, however, is more challenging. Since replicas are not trusted, our system can potentially suffer from forking attacks, where updates from different clients are hidden from each other [26]. We believe that database replication can provide the basis for a solution: different replicas can compare their local copies and thus detect divergence.

7 Conclusions

We have presented *Repeat and Compare*, a system for ensuring content integrity in peer-to-peer CDNs when replicas dynamically generate content. *Repeat and Compare* detects misbehaving replicas through attestation records and by leveraging the peer-to-peer network to repeat content generation on other replicas and then compare the results. Attestation records cryptographically bind replicas to their code, inputs, and dynamically generated output and build chains of accountability that help trace misbehavior. Our evaluation shows that *Repeat and Compare* is effective at quickly cleansing a CDN even if large fractions of replicas are misbehaving.

Acknowledgments

We thank Mike Freedman, Guy Lichtman, Lakshmi Subramanian, Jinyang Li and Dinh Nguyen Tran for their discussions. We also thank our shepherd, Barbara Liskov, and the anonymous reviewers for their feedback. This material is based upon work supported by the National Science Foundation under Grant No. CNS-0537252.

References

[1] A. Adelsbach, M. Rohe, and A.-R. Sadeghi. Towards multilaterally secure digital rights distribution infrastructures. In *Proc. 5th ACM DRM*, pp. 45–54, Nov. 2005.

[2] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. 20th SOSP*, pp. 45–58, Oct. 2005.

[3] S. S. Bakken, A. Aulbach, E. Schmid, J. Winstead, L. T. Wilson, R. Lefdorf, A. Zmievski, and J. Ahto. *PHP Manual*. PHP Documentation Group, Feb. 2004. `http://www.php.net/manual/`.

[4] R. J. Bayardo and J. Sorensen. Merkle tree authentication of HTTP responses. In *Proc. 14th WWW*, pp. 1182–1183, May 2005.

[5] L. Bent, M. Rabinovich, G. M. Voelker, and Z. Xiao. Characterization of a large web site population with implications for content delivery. In *Proc. 13th WWW*, pp. 522–533, May 2004.

[6] C. Canali, V. Cardellini, M. Colajanni, R. Lancellotti, and P. S. Yu. Cooperative archictectures and algorithms for discovery and transcoding of multi-version content. In *Proc. 8th IWCW*, Sept. 2003.

[7] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4):398–461, Nov. 2002.

[8] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[9] C.-H. Chi and Y. Wu. An XML-based data integrity service model for web intermediaries. In *Proc. 7th IWCW*, 2002.

[10] B. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. 3rd NSDI*, pp. 45–58, May 2006.

[11] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proc. 20th SOSP*, pp. 133–147, Oct. 2005.

[12] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th SOSP*, pp. 202–215, Oct. 2001.

[13] ECMA International. Common language infrastructure (CLI), 4th edition, June 2006.

[14] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693, IETF, Sept. 1999.

[15] M. J. Freedman. Personal Communication, Oct. 2006.

[16] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. 1st NSDI*, pp. 239–252, Mar. 2004.

[17] M. J. Freedman, K. Lakshminarayanan, and D. Mazières. OASIS: Anycast for any service. In *Proc. 3rd NSDI*, pp. 129–142, May 2006.

[18] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proc. 19th SOSP*, pp. 193–206, Oct. 2003.

[19] T. J. Giuli and M. Baker. Narses: A scalable, flow-based network simulator. Tech. Report arXiv:cs.PF/0211024, Stanford University, Nov. 2002.

[20] R. Grimm, G. Lichtman, N. Michalakis, A. Elliston, A. Kravetz, J. Miller, and S. Raza. Na Kika: Secure service execution and composition in an open edge-side computing network. In *Proc. 3rd NSDI*, pp. 169–182, May 2006.

[21] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proc. 2003 SIGCOMM*, pp. 381–394, Aug. 2003.

[22] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proc. 2nd IPTPS*, pp. 160–169, Feb. 2003.

[23] A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for byzantine fault detection. In *Proc. 2nd HotDep*, Nov. 2006.

[24] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proc. 21st PODC*, pp. 213–222, July 2002.

[25] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in P2P networks. In *Proc. 12th WWW*, pp. 640–651, May 2003.

[26] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. 6th OSDI*, pp. 121–136, Dec. 2004.

[27] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proc. 2nd NSDI*, pp. 99–114, May 2005.

[28] P. Maniatis, D. S. H. Rosenthal, M. Roussopoulos, M. Baker, T. J. Giuli, and Y. Muliadi. Preserving peer replicas by rate-limited sampled voting. In *Proc. 19th SOSP*, pp. 44–59, Oct. 2003.

[29] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *Proc. 1st Workshop on Internet Server Performance*, pp. 59–67, June 1998.

[30] OpenSSL. `http://www.openssl.org/`. Accessed Feb. 2007.

[31] H. K. Orman. Data integrity for mildly active content. *Proc. 3rd Workshop on Active Middleware Services*, p. 73, Aug. 2001.

[32] S. Pearson, B. Balacheff, L. Chen, D. Plaquin, and G. Proudler. *Trusted Computing Platforms: TCPA Technology In Context*. Prentice Hall, July 2002.

[33] Privoxy. `http://www.privoxy.org/`. Accessed Feb. 2007.

[34] B. Przydatek, D. Song, and A. Perrig. Sia: Secure information aggregation in sensor networks. In *Proc. 1st SenSys*, pp. 255–265, Nov. 2003.

[35] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proc. 20th SOSP*, pp. 235–248, Oct. 2005.

[36] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware '01*, pp. 329–350, Nov. 2001.

[37] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM TOCS*, 2(4):277–288, Nov. 1984.

[38] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proc. 20th SOSP*, pp. 1–16, Oct. 2005.

[39] E. Shi, A. Perrig, and L. V. Doorn. BIND: A fine-grained attestation service for secure distributed systems. *Proc. 2005 S&P*, pp. 154–168, May 2005.

[40] W. Shi, K. Shah, Y. Mao, and V. Chaudhary. Tuxedo: A peer-to-peer caching system. In *Proc. 2003 PDPTA*, pp. 981–987, June 2003.

[41] A. Shieh, D. Williams, E. Sirer, and F. Schneider. Nexus: A new operating system for trustworthy computing. In *20th SOSP Work-in-Progress Session*, Oct. 2005.

[42] E. G. Sirer. Meridian: Data Description, 2005. `http://www.cs.cornell.edu/People/egs/meridian/data.php`. Accessed Feb. 2007.

[43] Y. J. Song, V. Ramasubramanian, and E. G. Sirer. Optimal resource utilization in content distribution networks. Tech. Report CIS TR2005-2004, Cornell University, Nov. 2005.

[44] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. 2001 SIGCOMM*, pp. 149–160, Aug. 2001.

[45] K. Walsh and E. G. Sirer. Experience with a distributed object reputation system for peer-to-peer filesharing. In *Proc. 3rd NSDI*, pp. 1–14, May 2006.

[46] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *Proc. 5th OSDI*, pp. 345–360, Dec. 2002.

[47] Washington Post. The botnet trackers, February 16 2006. `http://www.washingtonpost.com/wp-dyn/content/article/2006/02/16/AR2006021601388.html`. Accessed Feb. 2007.

[48] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A lightweight network location service without virtual coordinates. In *Proc. 2005 SIGCOMM*, pp. 85–96, Aug. 2005.

[49] A. Yumerefendi and J. Chase. The Role of Accountability in Dependable Distributed Systems. In *Proc. 1st HotDep*, June 2005.

[50] A. R. Yumerefendi and J. S. Chase. Trust but verify: Accountability for network services. In *Proc. 11th ACM SIGOPS European Workshop*, p. 37, Sept. 2004.

[51] M. Yurkewych, B. N. Levine, and A. L. Rosenberg. On the cost-ineffectiveness of redundancy in commercial P2P computing. In *Proc. 12th CCS*, pp. 280–288, Nov. 2005.

[52] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J-SAC*, 22(1):41–53, Jan. 2004.

[53] W. Zhao and H. Schulzrinne. DotSlash: Providing dynamic scalability to web applications with on-demand distributed query result caching. Tech. Report CUCS-035-05, Columbia University, Sept. 2005.

[54] L. Zhou, F. B. Schneider, and R. V. Renesse. Coca: A secure distributed online certification authority. *ACM TOCS*, 20(4):329–368, Nov. 2002.