A Secure, Publisher-Centric Web Caching Infrastructure

Andy Myers[†] John Chuang[‡] Urs Hengartner[*] Yinglian Xie[*] Weiqiang Zhuang[*] Hui Zhang[*]
†Department of Electrical and Computer Engineering, Carnegie Mellon University
‡School of Information Management and Systems, University of California, Berkeley
∗Department of Computer Science, Carnegie Mellon University

Abstract—
The current web caching infrastructure, though it has a number of performance benefits for clients and network providers, does not meet publishers' requirements. We argue that to satisfy these requirements, caches should be enhanced in both the data and control planes. In the data plane, caches will dynamically generate content for clients by running code provided by publishers. In the control plane, caches will return logs of client accesses to publishers. In this paper, we introduce Gemini, a system which has both of these capabilities, and discuss two of its key components: security and incremental deployment. Since Gemini caches are deeply involved in content preparation and logging, ensuring that they perform correctly is vital. Traditional end-to-end security mechanisms are not sufficient to protect clients and publishers, so we introduce a new security model which consists of two pieces: an authorization mechanism and a verification mechanism. The former allows a publisher to authorize a set of caches to run its code and serve its content, while the latter allows clients and publishers to probabilistically verify that authorized caches are operating correctly. Because it is unrealistic to assume that Gemini caches will be deployed everywhere simultaneously, we have designed the system to be incrementally deployable and to coexist with legacy clients, caches, and servers. Finally, we describe our implementation of Gemini and present preliminary performance results.

I. Introduction

Web caching, like other forms of caching that occur at various levels of the memory hierarchy (e.g., hardware, operating system, application), exploits the reference locality principle to improve the cost and performance of data access. This has been especially effective at the Internet level, where large geographic and topological distances separate the producers and consumers of content. The direct and tangible benefits of web caching include: improved access latency, reduced bandwidth consumption, improved data availability, and reduced server load.

The main drawback of today's cache infrastructure is that it is network-centric, but not publisher-centric. From the publisher's point of view, a number of important features are missing. First, caches are not equipped to handle dynamically generated content, an increasingly large portion of all web traffic. Requests for dynamic content have to be forwarded back to the origin servers, and the dynamically constructed pages cannot be reused, even by the same client. Second, caches are unable to furnish reports on access statistics (e.g., hit counts and clickstreams) back to the publishers. This is of particular concern to publishers who rely on accurate hit counts to justify their advertisement-driven revenue model, and to publishers who wish to obtain accurate representations of the size and information consumption behavior of their audience. Finally, caches unilaterally make local copies of web objects, often without the consent or even the awareness of the publishers. Publishers have no knowledge of the number and locations of cached copies of their objects, making object consistency impossible to maintain. As a result, caches may be serving stale or outdated objects to the clients. For these reasons, many publishers have resorted to cache-busting, i.e., bypassing the caches by tagging their objects 'non-cacheable'. This forces the caches to forward all object requests back to the origin server. While this practice assures proper dynamic page generation, accurate hit counts, data consistency, and copyright protection, it also forfeits all the benefits of caching.

We believe that caching is fundamental to the long-term scalability of the web infrastructure, and therefore it is important to align the interests of publishers and cache operators. We propose a publisher-centric web caching infrastructure and paradigm that will encourage the publishers and cache operators to cooperate in the distribution and caching of web content. To accomplish this, we have built Gemini, a publisher-centric web cache and infrastructure.

The Gemini strategy is to endow cache nodes with communications, storage and processing capabilities that can be beneficially employed by publishers. A Gemini cache node is designed as a next-generation web cache that can be incrementally deployed in the current cache infrastructure. It can transparently substitute for a regular cache, as well as interoperate with existing cooperative caching schemes. A Gemini cache can support a variety of publisher-specified functions. In the data plane, it can support dynamic content generation using filtering, versioning, and/or other publisher-authored methods based on sandboxed languages such as Java. In the control plane, a Gemini cache can support customizable logging and reporting, as well as other functions such as object consistency control, access control, and publisher-specified QoS.

Central to our design is the architectural assumption of a heterogeneous global web cache infrastructure. Just as the Internet's routers and links are owned by different administrative domains, we assume that caches belong to many different administrative domains, and may have different functionalities. This assumption requires an emphasis on security mechanisms to protect publishers and clients from caches because (a) caches can transform content and (b) caches are not owned by a single organization that can be held accountable for any corrupted content. Also, our system must be incrementally deployable since there is no way to mandate that every domain must switch over to Gemini. In this paper, we address the problems of providing security and an incremental deployment strategy for Gemini.

The Gemini security architecture we introduce is designed to protect clients, publishers and caches from one another. Clients and publishers are assured of proper content generation and accurate logging by the cache, while caches are protected from malicious code from publishers. Our incremental deployment strategy allows Gemini caches and Gemini-aware servers to be gradually introduced without disturbing existing clients, servers, and legacy caches. Gemini caches automatically discover and use Gemini versions of documents, which legacy caches help to distribute.

The rest of the paper is organized as follows. Section II describes the different dynamic content generation techniques and applications supported by Gemini. Sections III and IV describe the security architecture and incremental deployment strategy. The design and prototype implementation of the Gemini node, based on the open source Squid [1] caching software, are presented in Section V. We discuss the performance of our implementation in Section VI, and identify related work in Section VII before we conclude the paper.

II. Gemini Applications

In this section, we give an overview of the potential set of applications that can benefit from Gemini. The purpose of this section is to argue by example that Gemini is relevant and useful. However, the focus of this paper is not on applications, but on the security and deployment issues associated with building Gemini.

Traditional caches can only handle static objects such as HTML pages. Gemini caches, on the other hand, are capable of storing and processing active documents, including the invocation of any publisher-authored methods based on sandboxed languages such as Java. This allows the Gemini caches to support, among a wide range of publisher-centric applications, the generation and delivery of dynamic content.

There are two main types of dynamic content. In the first case, a web page is dynamic because the underlying data source changes frequently. Examples include stock tickers, news headlines, and traffic reports. In the second case, a web page is dynamic because it is constructed on the fly on a per request basis. The exact form and substance of the page may be based on input from the client, server, and/or cache. Examples include database or search responses, customized news, customized page layout. Using a variety of techniques, Gemini caches can support both types of dynamic content generation.

When the underlying data source changes frequently, the cost-effectiveness of caching is dependent on the expected lifetime of the data. More specifically, the threshold for caching should be a function of the ratio of read-to-write frequency. For example, online stock tickers may be updated on a minute-to-minute basis, but popular tickers may be read multiple times per minute to justify caching. In many cases, a dynamic page consists of a small amount of frequently updated data embedded in a sea of static data. Instead of treating the entire page as uncacheable, a Gemini node can cache portions of a page according to publisher directives. Then it can generate new pages based on modular and differential page construction techniques. For example, delta encoding [2], [3] combines the data already in cache with any differential updates from the origin server. Other techniques include partial transfers, cache-based compaction [4], and HTML macros [5].

In the second case, Gemini supports on-the-fly page construction by running publisher-authored code for filtering and versioning, etc. Consider the application of filtering to the dynamic generation of customized news pages (e.g., MyYahoo). The publisher code residing at the Gemini cache will apply one or more filters to construct a customized page on the fly. The filters may be derived from several sources. First, filters may be supplied by the user in the form of cookies in the HTTP request header. For example, a user may specify the news categories and stock symbols that she wants to keep track of. Second, filters may be derived from a user profile match that incorporates her past browsing and purchasing history. This type of filter may be used for delivery of targeted ad banners, product recommendations and offers. Finally, the publisher code can generate its own filters by incorporating data that are specific to the local environment. For example, when the user accesses the URL from within her home area, the customized page may include local weather, traffic and sports news. When the user is travelling outside her home area, the page may include links to restaurants, accommodation, and maps for the foreign area instead.

Versioning is also useful for producing customized news pages. For example, a page may be laid out in different ways according to user-specified preferences stored in a cookie. The publisher code may also create different versions of the page for the same user based on the hardware device (e.g., desktop and handheld computers have different display capabilities), access bandwidth, operating system, and browser used to issue the request.

While we have used the example of a customized news page, these techniques can also be beneficially employed by other types of web sites. For example, a consumer e-commerce merchant may tailor web pages to individual customers with product recommendations, special offers, etc., based on the customer profile filter.

III. Security

In traditional distributed communication, end-to-end mechanisms are sufficient to secure data sent between the client and the publisher because intermediate nodes do not alter content. In our system, caches are active participants in content generation, so end-to-end security mechanisms are no longer sufficient. But it is not only dynamic content that affects the end-to-end nature of securing content delivery. Caches are now responsible for logging user hits as well. Publishers need assurances that caches will log accesses correctly, and that these logs will be transported back to the publisher intact. To accomplish this, caches must become fully involved in the system's security.

As an example, consider a publisher's digital signature on a document[1]. Previously, a client would be able to use the signature to verify the authenticity of the document. With Gemini, a cache between the publisher and client might transform the document according to a publisher's instructions, but the cache is unable to alter the publisher's signature because it does not possess the publisher's secret key. The result is that the client is unable to use the publisher's signature to verify the version of the document it receives. The obvious solution to this problem would be to distribute the publisher's secret key to caches, but this has serious ramifications: a cache with the secret key would be able to sign any content whatsoever on behalf of the publisher. Even if the cache's owner is honest enough not to exploit this, crackers who break into the cache may not be as polite.

Our design is guided by four principles:

Protect the publisher and client—not just the cache. Many previous systems have focused on protecting caches and clients from the publisher. However, it is just as important to protect the publisher and clients from caches.

The main risk to the publisher and client is of content being altered before it is delivered to the client. An attacking cache could edit or delete the publisher's objects, or add entirely new objects which appear to be from the publisher, so that what a client receives is not what the publisher intended to send. In addition to simply altering content, a cache could run a publisher's code incorrectly (either corrupting the program's state or the input given to the program).

Caches can also mishandle a publisher's content by prematurely ejecting it, by not respecting the quality of service that the publisher requested for the content, or by serving a stale version of the content. The first two of these affect performance but not correctness, while the last does affect correctness. Finally, a cache could add, alter, or delete entries from the log of client accesses recorded by the cache and returned to the publisher.

Publishers decide whom to trust. Like the other hardware in the Internet, we expect caches to be owned and administrated by a wide variety of organizations. We cannot expect every publisher to trust every organization, nor can we even expect all publishers to agree on which organizations are trustworthy. For this reason, we must let publishers decide which caches store and serve their content. Further, because some of a publisher's documents may be more valuable than others, publishers must be able to specify trust on a document by document basis. This allows a publisher to widely distribute less important content while keeping its most vital content in a smaller number of highly trusted caches.

Each publisher may choose to trust a set of caches, C, to serve a set of documents, D. Any cache not in C is not trusted to correctly run code from the objects in D. The publisher trusts that any cache which is a member of C will correctly store and run code from any object in D. Because there is a chance that a trusted cache will be compromised by an attacker, the publisher must still verify that trusted caches are functioning correctly.

A client will trust a publisher to deliver its own content. The client will also trust a publisher to delegate content delivery. Thus, if a publisher trusts some cache to deliver some document, the client will also trust that cache for that document. For the same reasons as the publisher, the client needs to verify that the cache is performing correctly.

A cache also trusts the publisher, and any cache the publisher trusts, to deliver the publisher's content. Other caches are completely untrusted, with one exception: a cache may trust other caches in the same administrative domain to deliver any document. Finally, even if it can be sure which publisher sent a piece of code, a cache will not trust that code to function correctly.

Publishers/clients find out about attacks eventually. If the publisher only trusts honest caches which are never compromised, its content will always be safe. But if trusted caches become dishonest, the system's security is endangered. Publishers and clients must have a way to detect these breaches of trust.

Many applications require that attacks be detected instantly, but in a caching infrastructure, instant detection is expensive because it requires the publisher to verify all content delivered. If we loosen the restriction and increase the amount of time an attack can go undiscovered, the cost of verification can be reduced since it can be done less often. Each publisher should be allowed to make its own decision about how long an attack can continue undiscovered since each publisher's content has a different value.

We believe that for most content, the value to a publisher of a single page being served correctly is very small. If a publisher's content is temporarily altered or made unavailable, the loss to the publisher is tolerably small. On the other hand, a publisher might wish to frequently verify content with a high value since even a short attack would have a significant cost. In this case, the high value of the content justifies a higher cost of verification.

The system should be incrementally deployable. The heterogeneous nature of the Internet prevents any system from being universally deployed in a short amount of time. Instead, new systems must be able to be deployed gradually,

[1] We consider a "document" to be a single object, rather than a whole "page," or group of objects which a browser might display together.

and must not interfere with existing systems. On the other hand, the system is not secure until the whole path from the publisher to the client is secured.
Neither caches nor client browsers should need to be modified. The caching infrastructure should be secure from the publisher all the way to the last cache or browser which has our system installed. Clients which do not run our system will be as vulnerable to attack as they are today since an attacker could alter content right before it arrives at the unmodified client.

The challenge in securing the cache is to come up with an approach that is both powerful enough to provide protection and generally applicable. For example, one approach would be to require that each cache include a secure coprocessor [6], which is a processor and memory encased in a secure, tamper-proof enclosure. The idea is that all parties can trust the coprocessor to oversee the generation of all content on the cache. Unfortunately, secure coprocessor technology is usually years behind commodity processor technology and more expensive due to the additional engineering and certification necessary to make the device tamper-proof. The resulting lack of performance makes a secure coprocessor unattractive for use in a web cache.

We employ two techniques to enforce the trust relationships outlined above: cache authorization and verification. The first is a way for publishers to explicitly specify which content a cache can generate. One key feature is that a client can determine that the content it receives is generated by an authorized cache. The client is the entity most interested in verifying that the content it receives has been produced by an authorized cache. And the client machine is often the least-contended-for resource on the path from the publisher.

Our second technique is a way for publishers and clients to verify that authorized caches are performing correctly. This allows the publisher to find out when a cache deemed trustworthy should not be trusted. We cannot prevent a cache from generating content or logging accesses incorrectly. Instead, we use non-repudiation of a cache's output to make establishing which cache is at fault easy. Coupled with random sampling techniques, any cache which misbehaves enough will be caught with high probability. Both publishers and clients can perform sampling to catch crooked caches.

Next, we present the details of our system. We begin by covering the authorization mechanism and security for content generation, which together address the first and second design principles. We then discuss our verification mechanism, which addresses the third design principle. Lastly, we consider the other side of the issue, describing how a cache can be protected from publishers. The fourth design principle, incremental deployment, is addressed in the design of all the mechanisms throughout this section.

A. Authorization

We rely on a public key infrastructure (PKI) to provide key distribution so that clients, caches, and publishers can check each other's digital signatures. There are several different PKI proposals [7], [8], but they all provide the basic service of associating a public key with an identity. This association is recorded in a certificate, which is a document signed by a certificate authority (CA). Each entity with a certificate can produce more certificates for other entities by acting as a CA. We assume there is a global root certificate authority which everyone knows and trusts.

Each publisher needs a certificate identifying its web site and public key. The format of the certificate is

$$\{P, K_P, Valid, Expires, CA\}_{K_{CA}^{-1}},$$

where P is the publisher's name and URL, K_P is the publisher's public key, from $Valid$ to $Expires$ is the range of time that the certificate is valid, and CA is the name of the certificate authority who created the certificate. The certificate is signed with the certificate authority's private key (K_{CA}^{-1}). Note that we also require each cache to have a public key and a certificate.

A publisher handles cache authorization decisions on an object-by-object basis. Each object includes an access control list (ACL) with which the publisher specifies which caches are allowed to store the object. The format of the ACL is

$$\{URL, K_1, K_2, ..., K_n, Valid, Expires, P\}_{K_P^{-1}},$$

where URL is the name of the object and each K_i is a public key. Each of the keys refers to a cache which is allowed to store the object. Instead of a list of public keys, the publisher can also specify a wildcard, indicating that any cache may process the current document. Observe that an ACL is just a special type of certificate, with the publisher acting as the CA.

A publisher can use entries in the ACL in two ways. One way is to authorize a single cache. This is accomplished by including a cache's public key in the ACL. The other way is to delegate the authorization decision to a third party. This is accomplished through a layer of indirection: the publisher includes the public key of the third party in the ACL. Then the third party creates certificates (signed with the key mentioned in the ACL) for caches it wishes to authorize. For example, a company such as Underwriters Laboratories might test caches for functionality and security and might issue certificates (signed by key $K_{ULapprove}$) for those models of caches that meet its criteria. The publisher could mention $K_{ULapprove}$ in its ACL if it trusts Underwriters Labs' judgment. A cache that has a certificate stating that it is a model that has been approved by Underwriters Labs would then be able to store the publisher's objects. As another example, consider an ISP with many caches. Assume the ISP uses public key, K_{ISP} to sign each of its caches' public keys. A publisher could mention all of the ISPs' caches as a group by including K_{ISP} in its ACL.

Altogether, a publisher would give the following to a cache: $ACL, \{Headers, Body\}_{K_P^{-1}}$. This is the ACL followed by the object itself. Note that the document and ACL are signed separately since the ACL will need to be passed on to the client. The $Headers$ field contains the URL and directives to the cache about how to handle the object (consistency, QoS parameters, log format, etc.). The $Body$ contains code and data which the cache uses to generate a reply to a client's request for the object.

Along with its response to the client's request, the cache includes the ACL. The client is able to check the signature on the ACL and use it to verify whether or not the cache is authorized to produce the requested object. If the cache is not authorized, the client should reject the document. Unauthorized caches are unable to convince the client that they are authorized.

B. Content generation

A cache's reply to the client has the following format:

$$ACL, \{URL, Cache, Client, H(Request), \\ CurrDate, Body\}_{K^{-1}_{Cache}}.$$

Except for the ACL, signed by the publisher, the cache signs the rest of the message: the URL requested, the cache's name, the client's name, a hash of any data sent in the request (e.g. for data conveyed in an HTTP POST message), the current date, and the requested content. There are three purposes for the cache's signature. First, it enables the client to detect tampering with the document on the path from the cache to the client. Second, it tells the client which cache generated the response. This enables the client to be sure that the author of the response is authorized by the publisher's ACL. And third, the cache's signature provides non-repudiation, linking the input (the URL and request data) to the output. The date and the client's name in the message serve to prevent replay attacks, where a third party sends a client stale data. In addition to the above information, the cache needs to provide the client with a chain of certificates which establishes that the cache is authorized by the publisher's ACL. This is done because the client cannot always determine which certificates are needed to link the cache's public key to one of the keys mentioned in the ACL.

One vital issue is how the cache can send this security information to the client in a manner that does not confuse legacy clients. Note that standard HTTP/1.1 [9] headers already contain the date, the cache name, and the URL. Further, the client computes the hash of the request itself. All the cache needs to send are the ACL, the signature, and certificates. We include these three items in the HTTP/1.1 Pragma header field. The HTTP specification states that clients and caches which are unable to parse this information will ignore it. Note that certificates are on the order of thousands of bytes in size. As an optimization, replies can contain the names of certificates rather than the certificates themselves. Certificates can then be cached to save bandwidth. Clients which do not implement Gemini security do not have to suffer the overhead of certificate transfer.

The client, after receiving the cache's response, needs to verify two things: that the cache is mentioned in the ACL, and that the cache's signature is valid. A valid signature implies that the content was not altered between the cache and client, and that both the client and cache agree on what the client's request was. If there is a problem in any aspect of the response, the client should discard it.

If the publisher desires, the cache can perform access control on the content using standard mechanisms such as username/password pairs, a cookie given to the client by the publisher, or according to the client's network address or hostname. If the publisher believes it is necessary, the cache can even require clients to access private data via SSL [10] or some other encryption layer. Standard SSL would not allow a cache to communicate on a publisher's behalf, but with the publisher's signed ACL, the client can be sure that the cache with which it is communicating is authorized by the publisher. Unlike our other mechanisms, allowing the cache to act as an SSL endpoint on behalf of the publisher requires modifications to the client.

C. Verification

Because a cache signs all of its responses to client requests, it is not able to later deny creating those responses. Any entity with access to the cache's certificate can verify the signature on a response. If a cache were to produce bogus content, its signature would be tantamount to a confession that it was the culprit. A client only needs to present the faulty output to the publisher to prove that the cache misbehaved. Once a publisher is convinced, it can remove that cache from all of its ACLs, preventing the cache from mishandling the publisher's documents in the future. The same technique works for catching a cache which fails to report log information. If a client presents the publisher with a signed response from a cache, the publisher can know to expect a log entry from the cache for that response. If the cache fails to return the log entry, the publisher knows that the cache is cheating.

The challenge is in determining when to question the cache's responses. We suggest two schemes: publisher-initiated auditing and client-initiated auditing. Both are based on random sampling so that the more a cache misbehaves, the higher the probability that it will be caught. In client-initiated auditing, the client sets a probability of verifying a cache's response with the publisher. After each response, the client flips a coin to determine whether to query the publisher.

Publisher-initiated auditing involves the publisher using a number of "fake" clients around the network to issue requests for the publisher's documents and return the responses to the publisher. Caches must not know which clients are fake so that the caches do not change their behavior when dealing with fake clients. Note that companies such as Keynote (http://www.keynote.com/) already have such clients set up to monitor web site performance. The publisher can then look at the responses to see that the cache has produced correct output. In addition, the publisher can verify that the fake client accesses were not over-reported or under-reported by caches, helping to assure the publisher that caches are performing logging correctly. In general, this technique cannot stop a cache from adding fake log entries by carefully inventing requests from non-existent clients. If a publisher is concerned about this attack, it can have a more trusted set of caches deliver some objects so it can be more confident about the logs returned. Another option is for the publisher to make some (small) object on a page uncacheable so that the publisher can handle all the logging itself while leaving most of the data distribution to the caching infrastructure.

Determining how much auditing should be done is a matter of trading network and cache resources for catching a misbehaving cache more quickly. As the sampling frequency is raised, caches are caught sooner but more system capacity is lost to the sampling process. Finding the right point on this continuum is beyond the scope of this paper.

D. Cache protection

The security mechanisms discussed so far deal with protecting clients and publishers from malicious caches. However, another concern is protecting caches from malicious attackers. For example, a publisher's code could attempt to access Gemini documents from other publishers or the underlying operating system. This problem is similar to the problem of protecting a web browser from malicious Java applet, so we adopt similar protection mechanisms. All of a publisher's code is run inside a sandboxed Java virtual machine so that a cache can have strict control over what operations the code is permitted to perform. In total, the API exposed to publisher code consists of functions for performing the following operations: read incoming request headers; write outgoing reply headers; write outgoing data; request arbitrary URLs to be loaded; and generate (a limited amount of) log info for each request. Another danger to caches is denial-of-service attacks, that is, code which consumes too many CPU cycles or allocates too much memory. To counter a denial of service attack, the quantity of CPU time, memory, and network bandwidth assigned to a publisher's code has to be limited. In our current prototype, we have implemented the API restrictions, but we have not implemented the resource limits.

IV. Incremental deployment

Having described Gemini's security architecture, we now present our deployment strategy. The Gemini infrastructure is designed to be incrementally deployable and fully interoperable with existing caches, servers, and clients. Gemini works with all types of cooperative caching, including hierarchical cache organizations. We have the following design principles:

Cache and document heterogeneity. Gemini caches co-exist and cooperate with legacy caches; not all documents have Gemini versions.

Transparency to clients. Clients need not be modified (except to achieve security).

Transparency to legacy caches. Legacy caches do not need to distinguish between Gemini and regular documents. They can fetch and cache Gemini documents, thereby assisting in their distribution. However, legacy caches will never serve Gemini documents to clients.

Proximity. Gemini content will be served by the authorized Gemini cache closest to the client, which we call the *leaf* cache.

Figure 1 shows an example caching hierarchy with a mixture of regular caches (X,Y,Z) and Gemini caches (G1, G2). For interoperability the publisher P will have two versions of its object, the regular version D and the Gemini version D'. Clients will request and receive D, not D'. In general, clients are never exposed to Gemini documents. Gemini caches are alerted by publisher P of the availability of D' (we describe the mechanism for this later in this section), but legacy caches can remain completely oblivious to the Gemini scheme and treat D and D' in identical fashion. Only Gemini caches (and publishers) understand the association between D and D'.

Let us illustrate Gemini caching by considering object requests by clients C1, C2 and C3 respectively. We assume all caches are initially empty, but Gemini caches G1 and G2 have been alerted to the availability of Gemini object D'. In response to a request for object D by C1, the Gemini cache G2 will perform a mapping from D to D' and issue a request for the Gemini version, D'. Caches G1 and X will forward the request back to P, and P returns D'. Caches X, G1, and G2 all make local copies of D'. Note that the legacy cache X does not know or care that D' is a Gemini object; it simply treats it as an opaque object. Now when D' arrives at cache G2, it is used to dynamically generate the object D for client C1. Next, client C2 issues a request for D. A cache miss occurs at the legacy cache Z, but a cache hit occurs at the Gemini cache G1. Since G1 received a request for D (not D') it knows it is the leaf cache. This is true in general: the cache which translates a request for a regular document into a request for a Gemini document will be the leaf cache. Therefore, it executes D' to dynamically generate D and sends it to C2. In this case, G2 may (at the publisher's request) make this copy of D non-cacheable by Z. Finally, C3 makes a request for D and it is propagated all the way back to the publisher. P will send the regular version D and may choose to mark the copy non-cacheable.

In the rest of this section, we describe how Gemini caches find out about the existence of Gemini documents. First, we present a method which utilizes the features of HTTP/1.1. Unfortunately, at least one widely deployed brand of web cache does not implement the features of HTTP on which this method relies. Therefore, we also present a second method which should work with every cache.

A. Discovering Gemini documents via HTTP

The HTTP/1.1 standard [9] allows a single URL to stand for multiple versions of the same document. This allows a server to offer a document in different languages (English,

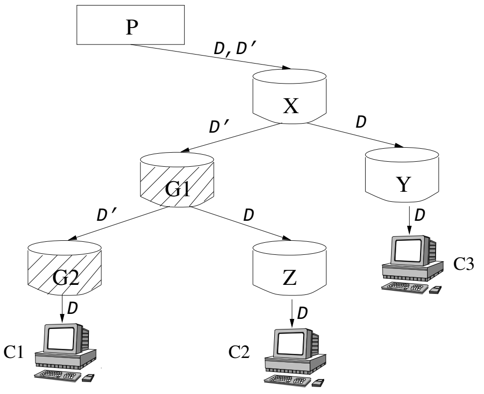

Fig. 1. Example caching hierarchy with Gemini caches (shown shaded). The regular version of the document is called D and the Gemini version is called D'.

French, or Russian), or with a different encoding (JPEG or GIF; compressed, gzip'd, or without compression). We can use this feature to distribute Gemini and legacy versions of a document by treating them as alternate encodings of the same URL.

When a Gemini cache forwards a request for a URL to another cache or the server, it adds a header which says that it can accept Gemini encodings. For example, a Gemini cache would add this line to say that it prefers the Gemini version of a document to any other encoding: `Accept-Encoding: gemini;q=1.0`. A server or cache which receives such a request can either reply with a regular version of the document or the Gemini version. If the reply is a Gemini version, then it will contain this header: `Content-Encoding: gemini`.

Note that a cache can store both the regular and Gemini versions of a document. The cache determines which version to send based on the request's `Accept-Encoding` header. Because a client will never specify that it accepts Gemini-encoded documents, it will never receive the Gemini version of a document. And since these headers are a standard part of HTTP, legacy caches can participate in Gemini document distribution without a problem. Unfortunately, some legacy caches ignore alternate encodings when selecting a document to return to a client, meaning that the cache could return a Gemini document to a client or cache not capable of handling the Gemini document. Thus, if our system is to be truly reverse-compatible, we must handle Gemini document discovery in a different way.

B. Second method for discovering Gemini documents

The approach we have chosen to implement is, in brief, to use different URLs for the legacy and Gemini versions of a document. Gemini caches are able to convert the URL of a regular document into the URL of the regular document's associated Gemini document. Legacy caches are unaware of the relationship between the Gemini and regular versions of a document, and merely send whichever one of the documents is requested.

There are two challenges to this approach. First, we require a robust way of transforming a regular document's URL into a Gemini document's URL. And second, we need a way for a server to notify Gemini caches about which of its documents have Gemini versions. One simple solution to these challenges would be to define URL naming conventions so that a regular document's name would indicate whether or not it had an associated Gemini document (and what the name of that Gemini document is). However, this approach is not robust. If a document's URL inadvertently contained the notation indicating it had an associated Gemini document, a Gemini cache might attempt to load the non-existent Gemini version. This could lead to increased delay for clients and additional useless requests for the server.

Our solution is to require the publisher to explicitly notify Gemini caches about which regular documents have associated Gemini documents. With each of its replies, a Gemini-aware server includes a notification in an HTTP Pragma header field. Legacy caches ignore the pragma, but Gemini caches store the publisher notifications as soft state. Each notification contains three pieces of information: a server name, a suffix to match, and a "tail," or string of characters, used to convert a regular document's URL into the associated Gemini document's URL.

When a request arrives, a Gemini cache looks at the URL. The cache finds all notifications with a server name the same as the server named by the URL. For each of these notifications, the cache tries to match the end of the path (the piece of the URL after the server name and port number) in the URL against the suffix in the notification. A match indicates that there is an associated Gemini document. On a match, the Gemini document's URL is formed by appending the tail to the URL in the request.

C. Authorization and leaf discovery

The authorization mechanism described in Section III complicates matters slightly when determining which cache is the leaf cache. Without authorization, the Gemini cache closest to the client will be the leaf. But with authorization, only the *authorized* Gemini cache closest to the client can be the leaf. When a cache receives a Gemini document, it can examine the ACL to determine whether it is authorized. But we can optimize this process slightly to avoid sending documents to caches which are unable to use them.

When a cache, C, initiates a request for a Gemini document, it also includes the names of its identification certificates. An upstream cache (or the publisher) will look at these certificates to determine whether C is authorized to serve the Gemini document. If it is, the document is returned. If not, an error message will be returned. Note that even if C lies about which certificates it possesses to acquire a document for which it is unauthorized, C still cannot produce a valid reply since it does not possess the proper keys.

D. Discussion

Two properties of our design make it especially scalable. By coexisting with the current caching infrastruc-

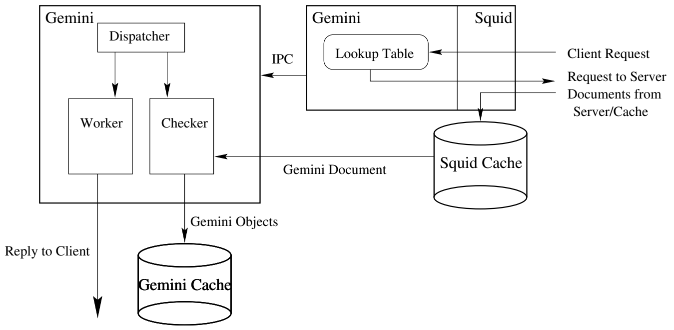

Fig. 2. Node Architecture.

ture, we are able to leverage thousands of legacy caches to help deliver Gemini documents. Also, observe that the leaf cache's task of producing a regular document from a Gemini document, which involves public key cryptography and possibly running code from the publisher, is a heavy-weight operation. We push this computational burden as close to the edge of the network as possible. Caches in the middle of the network, where resources are under the most contention, will usually only need to forward documents.

V. Node design and implementation

In this section, we discuss the design and implementation of a Gemini node. The performance of the implementation will be discussed in Section VI. We have implemented Gemini by extending an existing web cache, Squid. A block diagram of our system is shown in Figure 2. On the right is the Squid process, with three modifications: (i) a lookup table to store soft-state information on the availability of Gemini document versions, (ii) ability to fetch documents requested by the Gemini process, and (iii) forwarding of Gemini request (and document if applicable) to the Gemini process. On the left is the Gemini process, which includes a security module and a Java virtual machine (JVM). All code written by publishers is run inside the JVM. The security module is used to verify signatures on incoming documents and to sign outgoing documents.

A. Node operation

Now we describe the operation of the Gemini cache node, and explain the interactions between the various components shown in Figure 2. The Squid front end receives a document request, and in the event of a cache hit, satisfies the request immediately using the cache's copy to produce a reply for the client. In the event of a miss, it performs a table lookup (per Section IV-B) to query the existence of a Gemini version. If the Gemini version exists, and is cached locally, the Squid process will pass the request over to the Gemini process via IPC. Otherwise, Squid will initiate a fetch of the object using the standard caching hierarchy. When the object arrives, the Squid process stores the document in its cache and hands a pointer together with the original request to the Gemini process.

The Gemini process consists of three types of threads: a single dispatcher thread, a pool of checker threads, and a pool of worker threads. The dispatcher thread receives requests and documents from Squid and puts them into a request queue and a document queue, respectively, for subsequent processing. The checker threads are assigned to documents from the document queue, and they perform extraction of document parts and signature verifications. Depending on the Gemini document type, further processing is performed. Two types of Gemini documents are supported: active and non-active Gemini documents. Non-active Gemini documents are simply regular documents with appended signatures and the presence of some Gemini headers. These headers indicate, for example, the information to be logged when the document is requested. After checking the document's signature and parsing headers, non-active documents are stored by Gemini in its own cache. Active Gemini documents, on the other hand, may include Java classes in addition to headers and signatures. These classes are extracted and stored in a per-document directory.

The worker threads process the requests from the request queue. The JVM is invoked on the requests for active Gemini documents. It loads the Java class belonging to the document and runs it. The output is the document which will be sent to the client after being signed by the worker thread. The Java code may also create its own logging string. Alternatively, the standard Gemini logging facilities will log the request. If the code fails for a client request (due to programming errors, excessive resource consumption, etc.) the Gemini process will revert control of this request back to the Squid process. In this case the Squid process will handle the request as a regular document without a corresponding Gemini version.

VI. Performance evaluation

In this section, we present our preliminary performance results. Our main concern is in measuring the performance impact of our changes to the cache. We have conducted three experiments to quantify this impact: first, we measure the additional overhead on document lookups; second, we examine how long each stage of processing for a Gemini document takes; and third, we explore the overhead of a Gemini document without code (security is still in use, though). For all of these experiments, we use latency as our performance metric because we are interested in the potential response time degradation due to Gemini.

We have implemented Gemini on top of Squid version 2.2STABLE5 running on Linux (kernel version 2.2.13). For our Java virtual machine, we use IBM's Java Development Kit 1.1.8 with native threads. The server, cache, and client are each placed on separate machines (550 MHz Pentium III's with 128 MB of RAM) connected to the same 10BaseT Ethernet hub. All cryptographic algorithms are implemented using the Crypto++ [11] library (version 3.1). As an attempt to make the load imposed on the cache due to cryptography as light as possible, we use two different signature algorithms. The publisher signs its cer-

	Ad banner rotation		MyYahoo	
	Mean	(Std dev)	Mean	(Std dev)
Parsing	540	(18.5)	563	(22.9)
Extraction	821	(7.6)	5739	(184.6)
Sig. Check	2005	(10.7)	2079	(142.8)
Total	3926	(104.1)	8911	(173.8)

TABLE I
TIME TO UNPACK AN ACTIVE GEMINI DOCUMENT (μS).

tificates and documents using RSA [12], which has the property that signature verifications are fairly inexpensive. The cache signs all of its documents using DSA [13], with which signature production is fairly inexpensive. In all of the evaluation, the publisher issues a single certificate which explicitly authorizes the Gemini cache—no delegation of authorization is used. The publisher's certificate is appended to the Gemini document delivered to the cache. It is assumed that the client machine already has the publisher's certificate cached locally, so the cache sends only its signature along with the document.

A. Lookup overhead

As explained in Section IV-B, Gemini needs to search for a notification entry in a lookup table for each request received. Our first experiment is to determine the cost of this operation for regular documents without associated Gemini versions. We examine two cases: In the first case, there are no entries in the lookup table for the server named in the request. In the second case, there are lookup table entries for the server, but the request does not match the pattern specified by the entries. For example, a request might be for a URL ending in ".gif" but the entry's pattern only matches URLs ending in ".html". In both cases, it takes about 20μs for Gemini to perform the lookup operation. Compared with the normal hundreds of microseconds to tens of milliseconds required to process a document in an unmodified version of Squid, the penalty imposed by the lookup table is fairly small.

B. Active Gemini document

Our second experiment shows how long a request spends in each step during its processing. The processing consists of two stages: First, unless the document is already in the cache, the system has to download the requested Gemini document and to verify its integrity. Second, the reply for the request is generated by running the Java code in the Gemini document. We instrumented Gemini to timestamp the various processing steps, and we created 10 documents with identical content. To perform a measurement, we issue requests for all 10 documents, one after other. We repeat this procedure 10 times for 100 total requests. The very first request serves as warmup and is excluded from the results.

Tables I and II list the steps we are most interested in. We show both the mean and the standard deviation for each of them. They correspond to tasks of the major components of the Gemini process in Figure 2. Note that for all steps, the standard deviation is small when compared to the mean.

	Ad banner rotation		MyYahoo	
	Mean	(Std dev)	Mean	(Std dev)
IPC	144	(38.2)	144	(29.1)
Parsing	158	(31.1)	165	(30.0)
JVM	27580	(1847.6)	97210	(3682.8)
Signing	7364	(289.2)	7327	(268.9)
Total	37016	(1841.8)	106755	(3678.4)
Logging	178	(8.8)	230	(8.8)

TABLE II
TIME TO PROCESS A REQUEST FOR AN ACTIVE GEMINI DOCUMENT (μS).

We issue requests for two active Gemini documents, one containing simple code and the other containing complex code, in order to illustrate how code complexity affects node performance. The simple code (131 lines of Java) randomly inserts the URL of an advertising banner (from a static list of URLs) into a template HTML page. The complex code (559 lines of Java) generates a per-user, customized, MyYahoo-like page. Our implementation of this application is simpler than an actually deployed version would be: all of its required data is distributed in the Gemini document itself. In reality, certain data (e.g., stock quotes) would have to be dynamically downloaded by the Java code for each request and the Gemini document would contain only its HTML template. However, in our measurements, we are mainly interested in the time it takes to dynamically compose and sign a customized document and not in network latency. For this purpose, our simple implementation is sufficient.

Table I lists the various steps involved in unpacking a newly downloaded Gemini document. In the "Sig. Check" step, the system checks the signature attached to the Gemini document and verifies that the URL used for downloading the document matches the URL in the publisher's certificate. The overhead for this step is constant at about 2 ms, regardless of the type of Gemini document. Parsing of the reply headers also requires constant time for both documents, as opposed to the extraction step, which is much faster for the smaller Ad banner document (5 KB) than for the larger MyYahoo document (38 KB).

Table II shows the overhead of the single steps during the actual processing of the request. Running Java code and signing the freshly generated document are the most expensive steps. The running time for the code strongly depends on the type of active document: composing a MyYahoo-like page takes nearly four times as long as inserting URLs into a template page. The overhead for the signing operation is almost constant at about 7 ms because signing a document consists of computing a (cheap) constant-length hash over the whole the document and then performing (expensive) cryptography on the hash. Sending the request from Squid to Gemini via IPC and parsing of the request headers by Gemini also require a constant amount of time. The "Total" line corresponds to the time elapsed between the arrival of the request in the Squid process and sending the last byte of the reply by

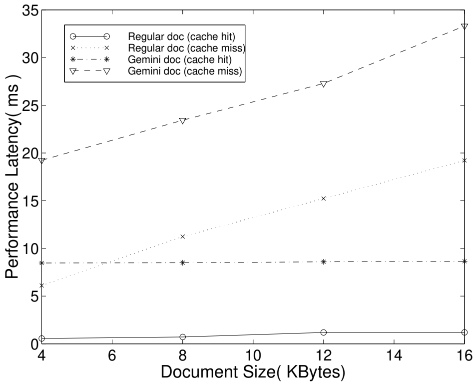

Fig. 3. Performance comparison of non-active Gemini documents and regular documents (ms).

the Gemini process. It does not include logging, since logging is performed after the reply has been sent. Logging is more expensive for the MyYahoo code than for the Ad banner code, since the former composes its own logging string, whereas the latter uses Gemini's default logging facility.

In general, by optimizing the security operations, (e.g., by using cryptography routines implemented in hardware), and by applying more-advanced Java techniques (e.g., compiling Java byte code to native code as soon as it is downloaded), we expect the performance penalties due to security and running Java code to decrease.

C. *Non-active Gemini document*

Our last experiment shows the latency increase from using non-active Gemini documents (Gemini documents without code) rather than regular documents. We evaluate four different document sizes from 4 KB up to 16 KB with an interval of 4 KB. The non-active Gemini documents are identical to the corresponding regular documents. In this experiment, we prepared 100 identical versions of each document, and then fetched these sequentially for 100 total requests. Figure 3 shows the average processing time for these requests. The first request serves as warmup and is excluded from the results.

From Figure 3, we can see that in the case of a cache hit when the document is already in the cache, the latency for non-active Gemini documents is about a constant time larger than that for the regular version. If we examine the times more closely, we find that the performance degradation is about 8 ms, among which signing the reply is the most expensive, accounting for 92% of the degradation. However, if there is a cache miss when the documents need to be fetched from the server first, the overall processing time degradation is not constant. While it takes a constant time to perform the security check, about 2 ms, the time spent on document extraction varies from 0.6 ms to 2.7 ms, depending on the document size.

VII. Related work

Related work comes from four areas: building distributed caching systems, web security, active systems (agents, networks, and caches), and research on securing active systems. There has been a large body of literature on web caching architectures (e.g. hierarchical and cooperative) and performance enhancement techniques (e.g. cache routing, push-cache). Gemini caches can work seamlessly in these caching architectures and most of the techniques are equally applicable to a Gemini-enhanced caching infrastructure. Within the last year, several private infrastructures such as Akamai [14], Adero [15], and Sandpiper [16] have been built to provide publisher-centric caching services. At the architectural level, the key difference between these systems and Gemini is that they assume all caches are under the same administrative domain, while Gemini assumes an environment where there are heterogeneous administrative domains and heterogeneous nodes (Gemini and non-Gemini). Because of this, Gemini has a strong emphasis on security and incremental deployment issues, which are not addressed in other systems. In addition, Gemini nodes support dynamic content, which, to the best of our knowledge, is not supported in these other systems.

There have been several efforts to bring increased security to the web. These include SSL [10], S-HTTP [17], [18], and the Digital Signature Initiative (DSig) [19]. All three of these protocols provide end-to-end security between the publisher and client, whereas the thrust of our work is in providing security even when a third party is generating content.

Gemini can be viewed as a special type of active network [20] with a focus on content delivery applications. Rather than making a router platform active, we make the cache platform active. In addition, we have a strong emphasis on trust and security issues, and discuss incremental deployment issues in the context of today's caching infrastructure. There are two other related active cache projects. Douglis et al. [5] proposes a highly specialized "macro" language which attempts to separate static and dynamic content in an HTML file. Basically, their scheme allows a cache to store some parts of an HTML file while fetching other parts from the publisher. In contrast, Gemini uses a general purpose language, Java, for data plane operations, and also, it allows publishers to specify control plane behavior. Cao et al. [21] have also enhanced a web cache with a Java runtime in order to allow caches to store dynamically-generated content. They emphasize the cache-centric features of their system: caches can choose which applets to store and how many resources an applet may take up. Further, their security model only considers protecting the cache. Our security model seeks to protect the publisher as well as the cache. Also, we give publishers more control over when and how their content is cached. Finally, we have considered how to deploy our solution in the existing caching infrastructure.

The problem of protecting active content from the host computer on which it is running has been explored, but comprehensive solutions have not been found yet. Moore [22] gives a good summary of work on software techniques and algorithms. The driving application for work in this area is on mobile agents. Work on securing the agent has focused on protecting state acquired at one server from being altered by other servers. In contrast, active content in our system does not alter its state as it moves from one cache to the next. Our main concern is that each cache should execute the code correctly. This is also a concern in the realm of mobile agents, but our problem is a somewhat easier one. The reason is that a publisher in our system is able to know what the output of each cache should be while the owner of an agent cannot know since the purpose of the agent is to gather previously unknown data. Yee [23] has proposed constructing a trusted, secure environment for mobile agents using tamper-proof hardware. Agents executing inside the environment can be sure that they will run without interference from malicious entities. This solution is generally applicable and we have considered using it in our own work. As we have said in Section III, the disadvantage of using trusted hardware is that its price/performance ratio is extremely unattractive due to engineering and construction costs. Building a high performance web cache using secure coprocessors would be prohibitively expensive.

The problem of protecting infrastructure from mobile code has been well studied. Moore [22] also contains a good survey of work in this area. Approaches come in several flavors: language-level protection and run-time checks [24] and load-time [25] checks. Our work builds on research from this area, applying mechanisms developed for mobile agents, or web browsers, to the domain of protecting caches.

VIII. Conclusion

We have introduced Gemini, an enhanced caching infrastructure which seeks to be publisher-centric. Publishers are given the ability to dictate how caches treat their content both in the data plane and the control plane. In the data plane, publishers are able to ship code to the caches to generate content dynamically. In the control plane, publishers can specify logging and QoS parameters. Learning from the success of the Internet's design, we adopt an architecture that allows caches with heterogeneous ownership and functionalities to coexist. To accommodate heterogeneity in an environment where content can be modified by caches, we present a security model which seeks to protect publishers from malicious caches using two methods: (i) giving the publisher control over which caches are authorized to generate content, and (ii) by providing verification mechanisms. In addition, we describe a deployment mechanism which enables Gemini to seamlessly interoperate with the existing caching infrastructure. We also present a node design which builds upon an existing cache, Squid, to implement Gemini's features.

Our preliminary performance evaluation shows that there are several areas in which Gemini could be optimized, especially in security. Future work includes implementing these optimizations as well as adding more features. For example, we wish to extend the publisher's control so that it can also specify the replacement policy for its content. And introducing additional data types such as streaming media would raise a number of research questions on topics ranging from quality of service to security.

References

[1] D. Wessels et al., "Squid web proxy cache," http://www.squid-cache.org.

[2] B. Housel and D. Lindquist, "Webexpress: A system for optimizing web browsing in a wireless environment," in *Proceedings of 2nd Annual International Conference on Mobile Computing and Networking*, 1996.

[3] F. Douglis G. Banga and M. Rabinovich, "Optimistic deltas for www latency reduction," in *Proceedings of 1997 USENIX Tech Conference*, 1997.

[4] M.C. Chan and T. Woo, "Cache-based compaction: A new technique for optimizing web transfer," in *Proceedings of IEEE INFOCOM'99*, 1999.

[5] F. Douglis, A. Haro, and M. Rabinovich, "HPP: HTML macro-preprocessing to support dynamic document caching," in *Proceedings of USENIX Symposium on Internet Technologies and Systems*, 1997.

[6] S.W. Smith and S.H. Weingart, "Building a high-performance, programmable secure coprocessor," *Computer Networks (Special Issue on Computer Network Security)*, pp. 831–860, April 1999.

[7] "CCITT, Recommendation X.509, The Directory–Authentication Framework," Consultation Committee, International Telephone and Telegraph, International Telecommunications Union, Geneva, 1989.

[8] R. L. Rivest and B. Lampson, "SDSI - a simple distributed security infrastructure," Available at http://theory.lcs.mit.edu/~rivest/sdsi11.html.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," IETF RFC 2616, June 1999, Available at http://www.ietf.org/rfc/rfc2616.txt.

[10] "SSL 3.0 specification," Available at http://home.netscape.com/eng/ssl3/index.html.

[11] Wei Dai, "Crypto++," Available from http://www.eskimo.com/~weidai/cryptlib.html.

[12] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 27, no. 2, February 1978.

[13] National Institute of Standards and Technology, *NIST FIPS PUB 185, "Digital Signature Standard"*, U.S. Department of Commerce, May 1994.

[14] "Akamai technologies, inc.," http://www.akamai.com.

[15] "Adero, inc.," http://www.adero.com.

[16] "Sandpiper," http://www.sandpiper.com.

[17] E. Rescorla and A. Schiffman, "The Secure HyperText Transfer Protocol," IETF RFC 2660, August 1999, Available at http://www.ietf.org/rfc/rfc2660.txt.

[18] E. Rscorla and A. Schiffman, "Security extensions for HTML," IETF RFC 2659, August 1999, Available at http://www.ietf.org/rfc/rfc2659.txt.

[19] Y. Chu, P. DesAutels, B. LaMacchia, and P. Lipp, "DSig 1.0 signature labels, using PICS 1.1 labels for digital signatures," *World Wide Web Journal*, vol. 2, no. 3, pp. 29–48, 1997.

[20] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," *Computer Communication Review*, pp. 5–17, April 1996.

[21] P. Cao, J. Zhang, and K. Beach, "Active cache: Caching dynamic contents on the web," in *Middleware '98*, 1998.

[22] J. T. Moore, "Mobile code security techniques," Tech. Rep. MS-CIS-98-28, Department of Computer and Information Science, University of Pennsylvania, 1998.

[23] B. S. Yee, "A sanctuary for mobile agents," Tech. Rep. CS97-537, University of California at San Diego, La Jolla, CA, April 1997.

[24] J. Gosling and H. McGilton, *The Java Lanugage Environment: A White Paper*, Sun Microsystems, Inc., 1996.

[25] G. C. Necula, "Proof-carrying code," in *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1997.