Harvesting Verifiable Challenges from Oblivious Online Sources

J. Alex Halderman
Princeton University
jhalderm@cs.princeton.edu

Brent Waters *
SRI International
bwaters@csl.sri.com

ABSTRACT

Several important security protocols require parties to perform computations based on random challenges. Traditionally, proving that the challenges were randomly chosen has required interactive communication among the parties or the existence of a trusted server. We offer an alternative solution where challenges are harvested from oblivious servers on the Internet. This paper describes a framework for deriving "harvested challenges" by mixing data from various pre-existing online sources. While individual sources may become predictable or fall under adversarial control, we provide a policy language that allows application developers to specify combinations of sources that meet their security needs. Participants can then convince each other that their challenges were formed freshly and in accordance with the policy. We present Combine, an open source implementation of our framework, and show how it can be applied to a variety of applications, including remote storage auditing and non-interactive client puzzles.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection (e.g., firewalls)*; E.3 [**Data**]: Data Encryption

General Terms

Design, Security

1. INTRODUCTION

In many distributed systems we want to verify claims made by remote parties. For example, in remote storage applications such as SafeStore [14], a client will pay remote servers to maintain a backup storage of his data. Such a client will want to check that servers that collect storage fees are actually using their resources to store his data. Elsewhere, in a peer-to-peer (P2P) systems, it is typically desirable that a malicious machine not be able to create several virtual identities and launch a Sybil attack. Thus, clients will want to verify that no machine owns several nodes.

Verifying such claims with certainty is often extremely costly or even impossible. For instance, if a remote server stores 100GB of data for a client, it is prohibitively expensive for the server to send all the stored data over the network each time the client wishes to audit the server. Fortunately, in practice it is often sufficient for a user to perform a probabilistic check of system claims. In a storage system one could query the server to send a randomly selected k blocks of data. A cheating server that only stored half of a client's data would have a vanishingly small (2^{-k}) probability of avoiding detection.[1] Similarly, a proof of work or client puzzle [10, 2, 12] used in a P2P system can be thought of a probabilistic "proof" that a certain amount of computational effort was spent by a machine solving the puzzle.

At the heart of these probabilistic checks is the idea that a random challenge is given to the party making a claim. The party will then need to create a response to this challenge such a returning a set of data or solving a client puzzle. In distributed systems we will usually want two properties from such a challenge/response mechanism. First, we want the challenge to be *fresh* such that the responder could not have predicted the response long in advance. Otherwise, a cheating server might only store the challenge blocks of data or a machine might pre-compute the solution to a client puzzle. Second, it is desirable that one response can be verified by several receivers. That is, the responder should not need to interact with and respond to each verifier independently. This property is important, for example, if several users in a P2P system want to verify a client puzzle, but we do not want to burden the puzzle solver with performing a separate computation for each of them. In addition, we would like to support systems where there isn't two way interaction between the verifier and responder, or where the identity of the verifiers isn't known in advance.

When prior, interactive communication with the verifier is not possible, the most straightforward solution is to deploy a set of trusted servers dedicated to generating challenges, but this approach has disadvantages of its own. Maintaining a secure dedicated server can be expensive, especially when

*This work was supported by the Department of Homeland Security Contract No. HSHQDC-07-C-00006, the U.S. Army Research Office Grant No. W911NF-06-1-0316, NSF CNS-0524252 and the US Army Research Office under the CyberTA Grant No. W911NF-06-1-0316.

CCS'07, October 29–November 2, 2007, Alexandria, Virginia, USA.

[1] Using redundant encoding, a user can expect that all of his data will be recoverable if such tests are met.

the number of participants in a system becomes large. While it would be convenient to rely on volunteer or peer-to-peer servers, it may be difficult to verify their legitimacy. Another risk is that dedicated random challenge servers might become targets of denial-of-service attacks. They may also be subjected to legal takedown measures if some of the system's users engage in illicit or politically sensitive activities.

Our Approach.

Rather than relying on dedicated trusted servers, we propose deriving what we call "harvested challenges" by mixing content from a variety of pre-existing online data sources, which can range from large news and stock quote providers to personal blogs. Using pre-existing data sources has several benefits. No new servers need to be deployed to provide challenges. The data providers that are harvested likely have infrastructure to handle large volumes of content requests, and since the primary purpose of their servers is unrelated to challenge generation, their removal for legal or political reasons would be much less likely.

While using existing content providers has advantages, our approach also presents a unique set of challenges. One problem is that we need a consistent method for deriving challenges from widely dissimilar types of content, none of which are tailored to our application. Another issue is that websites often modify or remove content after a period of time (for instance, many newspaper sites remove old stories). Finally, our scheme needs to be resilient against an attacker who can inject his own input as part of the challenge; for example, the adversary might post his own entry to a blog or alter the data on a compromised website.

Our solution to these problems is a general framework for deriving harvested challenges from Internet sources and an implementation of this framework called Combine (pronounced like the name of the harvesting machine, with the accent on the first syllable). Our system lets application developers decide how data from many sources should be combined to form challenges with an adequate level of trust for a particular use. Verifiers can later test that a challenge was properly derived in accordance with the application's policy by checking a subset of the original sources.

For example, an application's policy might require that challenges be generated by hashing together fresh content from six particular sites, and that verifiers must be able to confirm that the content of at least four of the sites was included in the challenge. This means the challenge will continue to be acceptable if any two of the sites modify their content or become unavailable. An attacker would need to compromise or predict four of the sources in order to precompute puzzle solutions based on this challenge.

Our framework is flexible in the types of sources that are harvested. For instance, certain client puzzle applications need to guarantee that challenges are derived from fresh content, a task for which many popular RSS feeds are well suited. On the other hand, cryptographic signature applications might be able to cope with less precise freshness but require that content be available for verification long after challenges are constructed. Policies might incorporate historical stock market data for this purpose.

Which combinations of sites to trust, how to balance between security and robustness, and the proper levels of freshness and long term stability are decisions that individual application developers will make to suit their particular requirements. Our framework allows systems that answer these questions in many different ways to create suitable challenges using a single tool, such as Combine.

Outline.

We begin in Section 2 by presenting an overview of our model and arguing about its security. In Section 3 we discuss how our approach can be applied to different security applications. Section 4 describes our harvested challenge framework, including our language for specifying data sources and policies. In Section 5 we introduce Combine, our implementation of the framework, and describe an email client puzzle application that we created to demonstrate its capabilities. Section 6 presents a study of RSS feeds that we conducted to evaluate the feasibility of our approach. We review several areas of related work in Section 7. Finally, we conclude and present ideas for future directions in Section 8.

2. HARVESTING CHALLENGES

In our setting there are two parties, which we will call the *deriver* and the *verifier* (see Figure 1). Our system attempts to achieve correctness and security properties with respect to these parties:

Correctness A properly behaving deriver is able to convince a verifier that the harvested challenge he presents was well formed in accordance with the application's policy. The system should be robust to some of the Internet sources becoming unavailable or removing or modifying their content.

Security Intuitively, for our system to be secure, we must prevent a corrupt deriver from being able to choose the derived challenge himself without the verifier detecting this. In addition, a deriver should not be able to leverage any precomputation he performed long before the challenge was derived.

We assume that the deriver and verifier both have access to Internet servers that will be used as sources of randomness. These sources may be unreliable, and fresh data might appear on them at irregular intervals. The deriver and verifier share a *policy* describing how challenges will be harvested and verified. In general, the policy will be chosen by the application developer and delivered along with the application. We assume that policies are known to potential attackers.

To construct a harvested challenge, the deriver first contacts n Internet sources specified by the policy and download some content from each. He then hashes each piece of content with a hash function H to produce outputs $h_1, \ldots, h_n$. Next, the deriver hashes the concatenation of all of these hashes to produce the harvested challenge u. The values $h_1, \ldots, h_n$ form a *derivation* for the challenge, which is communicated to the verifier.

Upon receiving the challenge and derivation, the verifier retrieves the content from the sources and verifies that their hashes match at least k of those given in the derivation, where k is a parameter that is given as part of the policy. Essentially, the verifier requires at least k of the sources to match the derivation and is willing to tolerate an error in the deriver's claim for the rest. By only requiring that a certain threshold of the sources match, the system is robust to the unavailability of some sources or the modification between derivation and verification of the content that they hold.

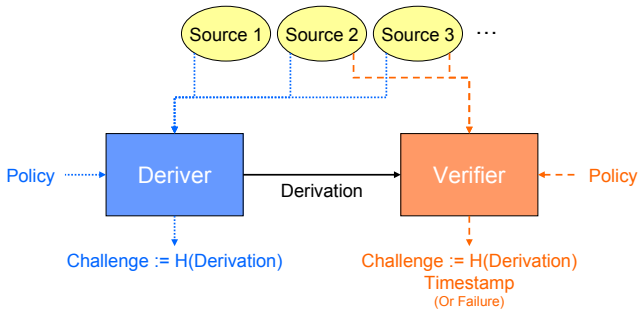


Figure 1: Our model involves an interaction between two parties, the deriver (blue/dotted lines) and the verifier (orange/dashed lines).

Intuitively, a larger k value will give a greater degree of security at the cost of making the system's correctness less robust to missing content. By selecting the kinds of sources and the values of n and k, application developers can create policies that strike a balance between the correctness and security requirements. Our framework also allows the construction of more complex policies, as described in Section 4.

Of course, unless the verifier can check that all of the content included in the derivation matches the content provided by the sources, a dishonest deriver will be able to replace some parts of the derivation with arbitrary values he selects. This turns out not to be problem for the applications we consider as long as the derivation still contains sufficient fresh entropy, since the harvested challenge is taken from the hash of the derivation.

For our purposes we will model H as an ideal hash function in the random oracle model [4]. Let x be the the number of sources that were both checked by the verifier and uncorrupted by a deriver. In addition, let t be the earliest time at which any of the content used from any of these sources was created and s be the amount of entropy from these x sources. Then if a corrupt deriver wishes to create a challenge u, he must create it by calling a random function that has as part of its input s bits of entropy out of his control. Therefore, if s is sufficiently large, say 80 bits[2], then the attacker needs to derive u by calling the ideal hash function. No amount of pre-computation before time t will help due to the fresh s bits of entropy.

3. APPLICATIONS

In this section we describe three application areas that can apply our challenge harvesting technique. We first present a storage application, then we give an example of a client puzzle system, and, finally, we introduce a novel approach to backward security of cryptographic signature schemes.

Remote Storage and Auditing Applications.

In the introduction we motivated our challenge harvesting problem with the application of auditing remote storage systems such as SafeStore [14]. Here we elaborate on a few details for how such a storage system might be built.

In a remote storage system a server will store data for a set of clients. The clients will want to periodically audit the server to verify that it continues to store their data. To respond[3] to an audit requirement, the server will derive a harvested challenge that specifies a random set of k blocks of data each of size b. The server responds by creating a message consisting of those k blocks along with the derivation sketch from our system. If a verifying client locally has a copy of the data that is backed up by the server the client can simply verify the derivation sketch and check the response blocks against his storage. In the case where the client does not maintain an independent copy of the data, we can apply a signature scheme to ensure integrity. When the data is originally created each block of data (along with its block number) will be signed by the data creator. The responses can then be modified to include the signatures on the challenge blocks. If signatures are of size s, then the extra storage overhead is s/b and the communication of a response is $k(s+b)$; this shows a trade off in storage efficiency and response efficiency in terms of the block size b.

Suppose that a storage server maintains a fraction x of the blocks data where $x < 1$. Then if the challenge set is of size k the server will have at most x^k chance of being able to respond to a particular challenge. For any significant data loss, the server's ability to respond correctly will decay exponentially in k. Data can be redundantly encoded such that only a certain fraction of the stored data needs to be recovered in order for the original data to be reconstructed. If we encoded the data such that half the stored data was required to reconstruct the original, then the system will detect unrecoverable loss of the *original* data with probability $1 - (1/2)^k$.

One way a server might try to cheat is to derive multiple random challenges until it finds one that it can satisfy. Suppose that the server derived r challenges, then by the union bound his probability of success is at most $r \cdot x^k$. In the random oracle model we can bound r by the number of calls made to the random oracle.

There are several other applications of using harvested challenges for auditing systems. For example, if a system was required to perform several database operations, a harvested challenge might be used to require that the transaction receipt be given for a certain set of them. One intriguing idea is to use harvested challenges in physical systems such as voting systems. For instance, we might use a harvested challenge to specify a certain group of precincts to perform a manual hand recount. Of course voting is an application of several subtleties and further analysis is required to determine the suitability of an approach like this.

Mitigating Sybil Attacks in P2P Networks.

Some of the most troublesome attacks to mitigate on peer-to-peer (P2P) networks are so-called Sybil attacks [9]. The survival of a P2P system often depends upon correct behavior of a majority of the participating nodes. In a Sybil attack, one machine will try to disrupt the system by claiming a large number of virtual identities. It will often be difficult to prevent this type of attack where there is no authoritative registration of users.

[2]Ideally, we would like s to be as large as the size of security parameters used in cryptographic systems. For example, several systems use 1024-bit RSA keys and 160-bit hash functions which roughly corresponds to "80-bit security."

[3]A response might be to an implicit request. For example, the server might be required to periodically broadcast a storage proof.

One technique that may be used to mitigate Sybil attacks is to require that a machine solve a client puzzle in order to gain an identity on a P2P system. A regular user will only need to solve a single puzzle, but an attacker must solve many in order to be effective. In this setting we have two conflicting goals. First, we want the client puzzle challenge to include some fresh randomness. Otherwise, an attacker could build up an increasing number of identities over time. Second, we want a user's proof of work to be verified by many other users, but the computational effort to gain an identity should be independent of this number (i.e., we don't want to require the user to provide a fresh proof of work for each verifier). Ideally, the user should not even need to interact with every party that verifies his puzzle solution.

We can use our framework to provide a straightforward solution to this problem. Let H_1 and H_2 be two independent hash functions that we will model as ideal hash functions. A node in a client puzzle system will first derive a harvested challenge u with appropriate freshness. Then, the node will choose random r until $H_1(u|r)$ satisfies some client puzzle condition (e.g., its first k bits are 0's). Then $H_2(u|r) = \text{ID}$ will be the user's identity in the system.

The harvested challenge was output from a random oracle that had a sufficient amount of entropy outside of the attacker's control. Therefore, precomputation before deriving u will not help an attacker. Although an attacker could always derive a different value of u by manipulating the inputs from the sources he has control over (or falsifying some parts of the derivation), this would only have the effect of calling the random oracle again and starting the client puzzle over.

Backward Secure Signatures.

In several applications of digital signatures a party will sign a message along with a timestamp denoting when it was signed. Forward secure signatures [3] are a method for evolving a private signature key forward in time such that if it is compromised at time t an attacker cannot use it to sign messages at an earlier time period t'.

Here we explore the opposite idea of preventing an attacker from being able to sign messages in the future if he compromises a machine at time t. At first this might seem impossible, since, if an attacker steals the private key material from a victim, there is nothing to prevent the attacker from posing as the victim himself. However, we show how to accomplish this in a more limited model known as the *bounded retrieval* model [6].

In the bounded retrieval model we assume that an attacker who compromises a machine will be limited in the amount of information he can steal from the machine due to constraints on his bandwidth or the possibility that exfiltrating large amounts of data will be detected. Previous work [6] under this model has suggested increasing the private key material in password systems so that an attacker who only manages to steal partial private key material will not be able to subvert the system.

Here we show how to build a signature system that prevents an attacker from forward dating a message. A user will create n private signing key/certificate pairs $(k_1, c_1), \ldots, (k_n, c_n)$ and store them on his machine. (The certificates can be signed with a signature key that is associated to the user and then discarded.) To sign a message M, at time t the user derives a harvested challenge u and hashes this to get a set, S, of k indices between 1 and n (where k is a system parameter). The user then signs M under all the private keys in the set S and also includes the certificates for the keys in S. A verifier will check that u was derived correctly and that all signatures verify.

If an attacker compromises a machine at time t and then subsequently leaves, he will only be able to sign with the keys he was able to retrieve. Therefore, if sufficiently many keys remain uncompromised at a future time t', the attacker will not be able to sign a message since he will not have all the keys dictated by the challenge u.

4. OUR FRAMEWORK

In this section we describe a generic policy framework for challenge harvesting that is suitable for a wide variety of applications with different freshness and security requirements. We believe this framework is simple enough to be used by application developers while flexible enough to adapt to a variety of data sources, including future ones that we have not anticipated.

4.1 Basic Operation

Tools that implement our system operate in two modes, which correspond to the role of the deriver and the role of the verifier, as shown in Figure 1. In deriver mode, the tool takes as input a *policy file*, which describes a set of content sources and a set of policies that specify what combinations of content from those sources will be acceptable to a verifier. The tool queries some of the sources and breaks the content from each source into *content chunks* based on the date and time when each piece of content was first posted at the source. When it has gathered enough content chunks to satisfy the policy, the tool packages them into a *derivation*, which it hashes with SHA-1 to derive the harvested challenge. The derivation is then passed to verifiers.

A derivation consists of a list of hex-encoded SHA-1 hashes of content chunks from each source, together with a timestamp (represented as an integer in the style of the UNIX `time` function) for each chunk that indicates when the data was first published according to the source. Time is an important aspect of both the derivation and verification processes. Policies may specify a maximum age for acceptable content chunks. Derivers interpret this age relative to the time when the derivation process began. This time (again, in UNIX format) is included as the first line of the derivation. Here is the format of a derivation:

```
derivation_timestamp

source1_name:
source1_chunk1_hash
source1_chunk1_timestamp
source1_chunk2_hash
source1_chunk2_timestamp
...

source2_name:
source2_chunk1_hash
source2_chunk1_timestamp
...
```

Optionally, a chunk timestamp may be followed by a comma and an source-specific data field used to verify the chunk hash.

Derivations contain the sources and chunks in a canonical

order so that the derived challenge can be defined as the SHA-1 hash of the derivation. Sources are ordered lexicographically by name, and content chunks within each source are ordered lexicographically by hash.

In verifier mode, the tool again takes a policy file as input, as well as a derivation created by a deriver. The tool inspects the derivation according to the policy and attempts to contact some of the content sources to verify a subset of the content chunks from the derivation. If enough of the chunks can be verified to satisfy the policy, the tool hashes the derivation to obtain the same challenge as the deriver. Otherwise, the tool returns an error. Verifiers interpret content age constraints as relative to the timestamp included in the derivation, and they output this timestamp along with the derived challenge if verification is successful. Applications can use this timestamp to decide whether the challenge is fresh enough for their purposes.

The policy files used in our framework are specified with a policy language that we introduced later in this section. (A Python YACC grammar is included with Combine and should be considered the authoritative description.)

4.2 Data Sources

Our framework abstracts all types of content providers into a notion of a *source*. Sources can encapsulate many different kinds of data, such as news stories, stock quotes, web pages, and blog entries, delivered in different kinds of formats, such as HTML, RSS, and other XML schemas.

Policy files define sources using the following syntax:

```
source source_name (
  type = source_type
  attribute = "string_value"
  attribute = numeric_value
  ...
)
```

A source's `type` attribute refers to a source handler module within the tool, and all other attributes specified for the source are passed directly to that module. Each module oversees the way source data is retrieved, divided into content chunks, cached, used in challenge derivation, packaged into a derivation, and applied to challenge verification. This level of abstraction means the framework can be extended to support many other kinds of sources. In this section we describe the sources we have implemented in Combine.

RSS Feeds.

RSS feeds [17] are well suited as a source for harvested challenges. There are many feeds available—the aggregation site Technorati tracks about 69 million [18]—and they represent many kinds of content, from newswire stories to personal journals entries, in a consistent data format. An RSS feed typically includes the most recent 10–50 pieces of content posted to the site providing the feed, so tools can retrieve several content chunks with a single HTTP request. Each content item includes the date when it was posted or last updated, which helps tools ensure the freshness of derived challenges.[4] Sites that provide popular RSS feeds are equipped to serve a large volume of subscribers whose RSS reader software refreshes the content at regular intervals, so they are likely to have the capacity to absorb extra load from being used as a challenge source by a smaller number of users.

Despite these advantages, RSS feeds also pose specific difficulties for our purposes. Sites publish new entries to their RSS feed at widely varying rates—ranging from many times an hour to once every few months. Since feeds only contain the most recent entries, if they are updated too quickly, content may age out of the feed so fast that verifiers will be unable to check them, even though they would otherwise be fresh enough to meet the application's needs. On the other hand, if entries are published too infrequently, derivers may not be able to collect enough recent content to satisfy the policy. A similar difficulty stems from the tendency for sites to update items that have already been published to a feed, replacing the old content. If many of the entries in an RSS feed change between derivation and verification, the policy may not be satisfied.

Clearly, not every RSS feed will be a suitable challenge source for every application, but developers can select specific feeds based on their past publication rate and tendency to replace entries. Applications that require fine-grained freshness over long-term verifiability should select feeds that are published frequently, while ones where challenges from longer in the past need to be verified should choose feeds that are updated less frequently. We have conducted empirical tests of RSS publication and update frequency that suggest there are many feeds suitable to each kind of application. These results are reported in Section 6.

RSS feeds are represented in policy files as sources with the following attributes:

```
source source_name (
  type = RSSFeed
  url = "url"
  min_entries = m   (optional, default 1)
  max_entries = n   (optional, default ∞)
  max_age = t       (optional, default ∞)
)
```

The `url` attribute specifies the feed location. Derivers will include up to `max_entries` content chunks from the feed in derivations. Only chunks that are up to `max_age` seconds old will be used. Verifiers will also check that at least `min_entries` of them match entries still present in the feed and unmodified when the derivation is verified. They will check that the entries are dated no more than `max_age` seconds before the derivation's timestamp.

For example, this source represents an RSS feed provided by a national newspaper:

```
source NYTimes (
  type = RSSFeed
  url = "http://www.nytimes.com/services/xml/rss/-
    nyt/HomePage.xml"
  min_entries = 5
  max_entries = 20
  max_age = 86400
)
```

Here, derivers will include up to 20 entries from the past 24 hours and verifiers require at least 5 to match.

Historical Stock Market Data.

Stock market data has been proposed as a source of public randomness because it is widely disseminated from reputable sources and difficult to exactly predict [20]. (An adversary

[4]These timestamps can sometimes be wrong, but applications that depend on freshness should create policies that incorporate a number of sites that have a history of maintaining accurate clocks.

who can accurately predict stock prices is likely to have more lucrative uses for this skill than guessing harvested challenges.) Furthermore, online sources provide daily historical share prices extending years into the past. Such data is suitable for constructing challenges with time granularity to within one trading day that can be verified long after they are derived.

Historical market data sources fit neatly within our framework. Policy files define them with the following syntax:

```
source source_name (
  type = DailyQuotes
  symbols = "ticker_symbols"
  min_entries = n   (optional, default 1)
)
```

The deriver creates one content chunk for each quote using data from the last full trading day. Each chunk consists of the SHA-1 hash of the concatenation of the symbol, the date, the opening, high, low, and closing share prices, and the trading volume. Consistent data can be retrieved from several different online sources.

For example:

```
source TechStocks (
  type = DailyQuotes
  symbols = "GOOG,YHOO,MSFT,INTC,IBM"
  min_entries = 4
)
```

Here, derivers will include content chunks for up to all five stocks, and verifiers will require at least four of them to match figures obtained from a trusted historical market data provider for the last full trading day prior to the date of the derivation.

Explicit Randomness Servers.

Many traditional client puzzle schemes have relied on explicit servers to issue random challenges. For some applications, it may be appropriate to use an explicit server as the main challenge source, but to use challenges harvested from oblivious online sources as a backup in case the server is unavailable. Our framework can accommodate dedicated challenge servers using the `RandomServer` source type. This allows applications to create policies that use explicit randomness but fail over to derived randomness for added robustness.

An explicit random server is defined like this:

```
source AppsRandomServer (
  type = RandomServer
  url = "url"
  max_age = t
  verify_key = "filename"
)
```

When the tool derives a challenge from this source, it makes an HTTP request to `url`. The server responds with three lines: a 160-bit hex-encoded pseudorandom value, a UNIX-format timestamp, and a DSA signature of the first two lines that can be verified with `verify_key`. The deriver uses the pseudorandom value and timestamp as the sole content chunk for the server. The derivation also includes the signature as the optional verification value. When verifying the challenge, the source is considered valid if the signature can be verified with `verify_key` and the signed timestamp is no more than `max_age` seconds older than the derivation. Thus, the content chunk can be verified without further interaction with the server.

Other Sources of Randomness.

Web sites provide many other sources of randomness that could be utilized within our framework. A viable source must contain sufficient fresh entropy that is at least partially beyond adversarial control and can be accessed by verifiers after the challenge is created. One potentially useful class of sources is sensor data, such as archived feeds from earthquake [16] and sunspot [15] monitoring systems. These may provide better stability than RSS feeds, since data from them will rarely change after it has been published. Another promising category includes change logs from frequently edited Wikis and source code repositories. Although we have not yet implemented these sources in our Combine tool, future versions easily could accommodate them within the existing policy framework.

4.3 Policy Descriptions

We designed our policy language to cope with a variety of unreliable data sources. Some sources may be unreachable when a challenge is being derived, others may change or become unavailable by the time the challenge is verified. Like sources, policies may define minimum and maximum numbers of sources from different sets. This instructs derivers to include additional sources, up to the maximum, for increased robustness at verification time.

Policy files contain one or more policy definitions. A simple policy takes the form:

```
policy policy_name { source_1, source_2, ...}
```

For example:

```
policy PickOne { NYTimes, CNN, Slashdot }
```

By default, the tool assumes that at least one of the sources must be satisfied, but it will attempt to include content chunks satisfying all of them in the derivation. A policy can override this by specifying a minimum number of sources to verify and maximum number to include at derivation, like so:

```
policy policy_name { source_1, ...}[min,max]
```

For example:

```
policy PickTwo { NYTimes, CNN, Slashdot }[2,2]
```

During derivation, the tool will include content chunks from up to two of these three source, and verifiers will require any two to be satisfied. The tool evaluates the sources in order from left to right, so content chunks from `Slashdot` will not be used unless the deriver cannot retrieve the minimum number of content chunks required from either `NYTimes` or `CNN`.

Sometimes when a source is referenced within a policy, we may want to use different criteria for determining when it is satisfied than those we specified when the source was defined. To do this, we can specify local source attributes that override the source's global attributes. For example:

```
policy { NYTimes(min_entries=10) }
```

Policies may also be nested. In this example, an explicit random server is used preferentially during derivation, but a set of RSS feeds will be used instead if the server is unreachable:

```
policy FailOver { AppsRandomServer, { NYTimes, CNN,
  Slashdot }[2,3] }[1,1]
```

A policy file can specify multiple policies, and they can

refer to each other by name in place of sources, as in this example:

```
policy Nested {
  { NYTimes, CNN, Slashdot }[2,3]
  Recent
}
policy Recent {
  NYTimes(min_entries=1, max_age=3600)
  CNN(min_entries=1, max_age=3600)
}[2,2]
```

For this policy to be satisfied, the verifier needs to validate the default number of sources from two of the three RSS feeds, or, at least one story from within the past hour from each of `NYTimes` and `CNN`.

5. IMPLEMENTING COMBINE

In this section we describe Combine, our implementation of the framework introduced in the previous section. We have released Combine under a BSD-style license, and we invite application developers to adopt it. The source is available at `http://www.cs.princeton.edu/~jhalderm/projects/combine/`.

We wrote Combine in Python because of the language's multi-platform support, object-oriented structure, and robust tools for parsing RSS feeds and other Internet data sources. This simplified the tool's construction significantly, but we have been careful not to define any of framework in terms of Python-specific behavior, so as not to preclude interoperable implementations in other languages. Combine provides a simple API that can be called from other Python applications as well as a command line interface suitable for use with a variety of other languages and scripting tools.

Combine supports the three kinds of data sources detailed in the previous section: RSS feeds, historical market data, and explicit challenge servers. It interprets RSS feeds using the Python Universal Feed Parser, which supports the most popular versions of the RSS and ATOM feed specifications.

Combine caches feed data on the client in order to minimize redundant requests. This reduces the risk that an attacker could execute a denial-of-service attack on the sources by causing many challenges to be generated. Combine also obeys the robots exclusion standard [13], so content providers can choose not to be available for harvesting if they desire.

5.1 Complete Example

To better illustrate our system's behavior, we now give an end-to-end example of Combine in use. Here is an example policy file (stored in a file named `example.pol`) that defines a policy similar to the one shown at the end of the preceding section:

```
source NYTimes (
  type = RSSFeed
  url = "http://www.nytimes.com/services/xml/rss/-
    nyt/HomePage.xml"
  min_entries = 5
  max_entries = 10
  max_age = 86400
)

source CNN (
  type = RSSFeed
  url = "http://rss.cnn.com/rss/cnn_topstories.rss"
  min_entries = 5
  max_entries = 10
  max_age = 86400
)

policy Example { {NYTimes, CNN}[2,2], Recent }
policy Recent {
  NYTimes(min_entries=1, max_age=3600)
  CNN(min_entries=1, max_age=3600)
}
```

This policy file defines two sources, both of which are RSS feeds from online news providers. It also defines two policies. One policy is named `Recent` and requires one verifiable content chunk made from a story from either source but no more than an hour old when the challenge was derived. The other policy, called `Example`, is considered valid either if both sources are satisfied with their default arguments (a minimum of 5 entries from each, no more than 24 hours old), or if `Recent` is satisfied.

Suppose Alice wants to derive a challenge from this policy and output a derivation to a file named `alice.d`. She invokes Combine like this:

```
$ combiner -policyfile example.pol -derivation \
    alice.d -derive
```

By default, Combine will apply the first policy defined in the policy file, so it will apply `Example` here. This behavior can be overridden using the `-policyname` argument.

Assuming that the tool can find enough fresh RSS entries to satisfy the policy, it outputs a message that indicates success:

```
derived:  Example, a936b92d6497..., 1169960994
```

This status line contains the name of the policy that was applied, the derived challenge, and the challenge's timestamp (which is normally the time when the derivation process started, though it can be backdated with the `-time` argument).

Alice can now use the challenge as the basis for a client puzzle. She solves the puzzle, and sends the solution to Bob along with the derivation returned by the tool.

The derivation looks like this:

```
1169960994

CNN
9a35a2442faf16b994f10b75573a18269fa4b97f
1169942568
a33524271ed0d5e9b0d2aafa8c0ed2a6c39a1b78
1169956546
(... 6 more content chunks)

NYTimes
8829863ca791800d164b546d03503eb74294713f
1169959720
d7190efb9603e0d88ea2cdff0134c5416ecfd656
1169959401
(... 9 more content chunks)
```

Note that `CNN` has only 8 content chunks—fewer than the 10 chunk limit specified by the source's `max_entries` parameter. This indicates that the RSS feed only contained 8 entries that were younger than `max_age`. Also notice that though each source was referenced twice in the policy, there is only one section for each source in the derivation. The reason is that the framework specifies that there is one derivation section for each source name, so the tool collapses all content chunks from each into a single section. This pool of content chunks will be consulted any time the source is mentioned by the policy during verification.

A few minutes later, Bob wants to verify Alice's derived challenge. He runs Combine using the same policy file and the derivation that Alice sent:

```
$ combiner -policyfile example.pol -derivation \
    alice.d -verify
```

Suppose that a network outage is preventing Bob's computer from reaching the CNN web site to retrieve its RSS feed. Can the `Example` policy still be satisfied? The `{NYTimes, CNN}[2,2]` subpolicy is not satisfied, since it requires some content chunks from both sources to be validated. However, `Recent` can be satisfied. Notice that the first content chunk from `NYTimes` was only 1274 seconds old when the challenge was derived. `Recent` only requires one content chunk from either source to be verifiable as long as that content chunk was less than an hour old at the time of derivation, so the `Recent` subpolicy will be satisfied as long as the RSS feed entry corresponding to this content chunk is still present and unmodified when Combine checks the feed. If so, `Example` will be satisfied, and the tool will output this message:

```
verified:  Example, a936b92d6497..., 1169960994
```

Now Bob can check the challenge's timestamp to ensure that it is fresh enough for his purposes, and he can check that Alice's solution is valid for the derived challenge.

5.2 Anti-Spam Demonstration

To demonstrate our tool's capabilities, we have created a prototype anti-spam application called Postmark that uses derived challenges from Combine to improve upon existing schemes.

One limitation of traditional proof-of-work anti-spam systems like Hashcash [2] is that they do not guarantee the freshness of their random challenges. This allows an attacker to perform a large amount of work ahead of time and then send burst of messages in a short period.

Our Postmark application uses hash puzzles based on fresh challenges derived by Combine to limit an attacker's ability to precompute puzzle solutions. Combine allows the system to ensure freshness without having to interact with the receiver prior to sending a message or to rely on a dedicated challenge server.

Postmark consists of two modules, an SMTP proxy server, `postserv` and a Procmail-compatible message filter, `postcheck`. Both are implemented in Python and incorporate Combine using its API.

Postmark uses hash puzzles that are formed by concatenating and SHA-1 hashing the message body (along with user-visible headers such as the sender, receiver, subject, and date), derived challenge, and receiver envelope address, as follows:

$$p = H(H(message\_body)|derived\_challenge|receiver\_address)$$

For a difficulty parameter n, a solution to the puzzle is a string s such that $H(s|p)$ has n leading zeroes. Postmark solves puzzles with a separate process written in C that uses a highly optimized SHA-1 implementation. Of course, verifications are nearly instantaneous after the derived challenge is verified, since they require only two additional hash computations.

The puzzle difficulty, the policy used to harvest the challenge, and the maximum age of acceptable challenges are configurable. However, these parameters would need to be standardized prior to widespread deployment to ensure interoperability. We have tested the application with a sample policy like the one in the previous section, using puzzles that take approximately 10 seconds to solve on a fast machine ($n = 25$) and are considered fresh for up to 60 minutes.

The Postmark server application acts as a local SMTP proxy. It receives messages sent from an unmodified SMTP client running on the local machine and appends puzzle solutions before relaying them to the user's normal SMTP server. This design allows the tool to be nearly transparent to the user, requiring only a simple mail client configuration change.

When the Postmark proxy server receives a message, it uses Combine to derive a fresh harvested challenge, then solves a client puzzle based on the challenge. It wraps the original message in the MIME `multipart/signed` message type [11] (this directs mail transfer agents not to modify the body in transit and assures that the Postmark data will not cause problem for other receivers). It then attaches a second message part containing the following values: the receiver address, the derived challenge, s, n, and the derivation created by Combine. Finally, it passes the message to the original SMTP server for delivery.

Recipients invoke the Postmark filter on incoming mail using Procmail or another message filtering system. The Postmark filter reads the received message from standard input and tries to detect a Postmark puzzle solution. If one is found, the filter checks that the recipient address used in the puzzle matches the local user's address, it uses Combine to verify the challenge, it uses the timestamp on the challenge to check that it is suitably fresh, and it verifies the puzzle solution. If all these tests are passed, the Postmark filter returns an exit code of 0; otherwise, it returns a non-zero exit code. The user's Procmail system can be configured to allow mail with a valid Postmark to bypass other spam filters or receive less scrutiny than unmarked mail.

6. EVALUATION: RSS FEEDS

RSS feeds show considerable variation in several respects that are important for our purposes. They differ in terms of how many entries are served at once, how often new entries are posted, how long entries remain available for verification, etc. We conducted an empirical study to determine whether actual RSS feeds are suitable for use with our system.

We studied two sets of sources. The first consists of the most widely subscribed feeds as reported by the Bloglines feed aggregation service on January 10, 2007. This set contains 133 feeds, and we label it "Popular." The second set consists mainly of more esoteric feeds, which we compiled by combining the subscriptions of four members of the Princeton computer science department and removing any sites in the Popular set. We believe this list is somewhat representative of the majority of RSS feeds, which fall into the long tail of the popularity distribution. This set contains 142 feeds, and we label it "Longtail." For both data sets, we requested the contents of each feed once every 10 minutes over a seven day period in January 2007.

Availability.

Applications that require robustness will need to select RSS sources with a high likelihood of being available when challenges are derived and verified. We considered a source to be available if it was reachable and returned parsable

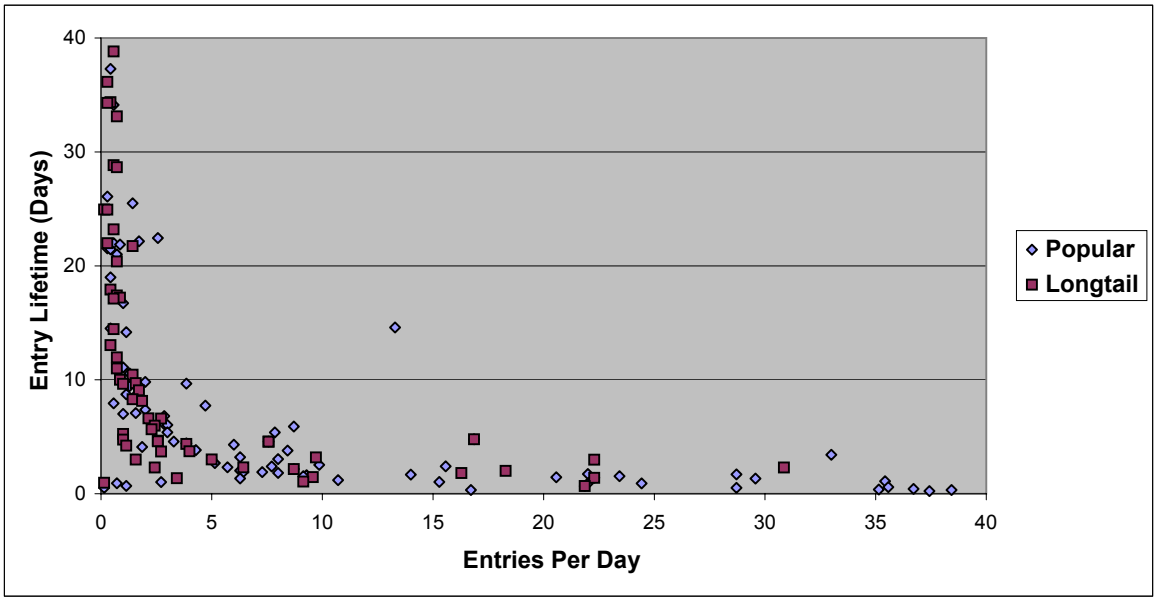


Figure 2: Comparing the frequency of new posts and the average time posts remain in the feed reveals the expected inverse relation.

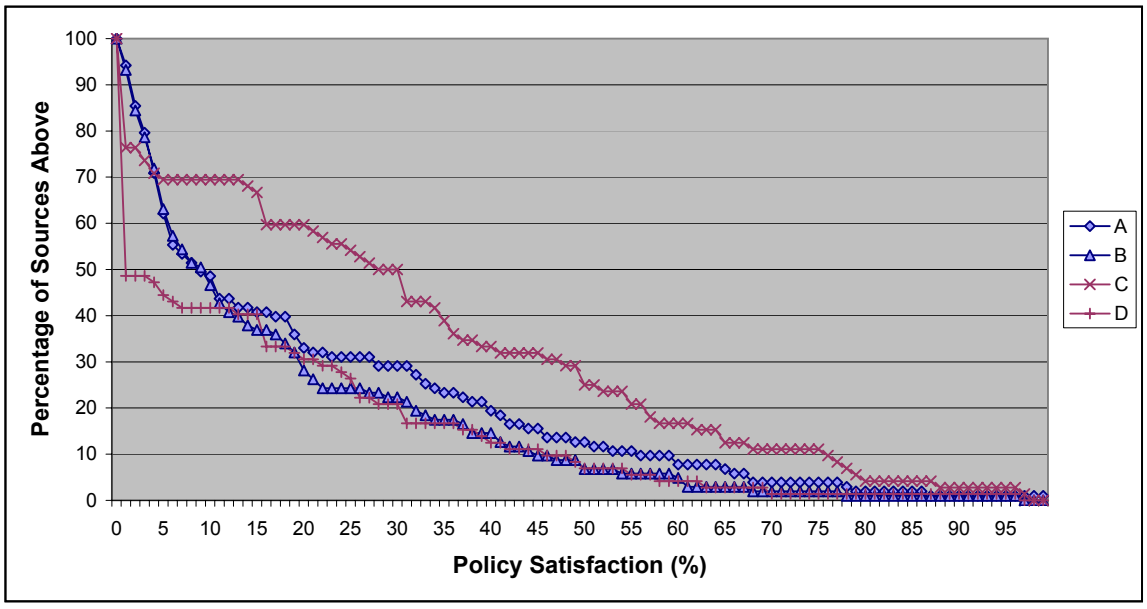


Figure 3: We modeled simple policies and robustness requirements using feed data collected from our sources and recorded the portion of the study period when each policy was satisfied. Here we plot the percentage of sources that met or exceeded each level of satisfaction. The scenarios are: derivation using all posts fresh within one hour, at least one post verifiable up to (A) 6 hours later, (B) 12 hours later; derivation using all posts fresh within one day, at least one post verifiable up to (C) 7 days later, (D) 14 days later.

entries at a sample time. We disregarded sites that were never available during out study (this was usually caused by an outdated URL or chronically malformed data). For the Popular data set, 2 of the 133 feeds never returned usable data and were discarded. Of the remaining feeds, 90% were available at least 99% of the time, and 97% were available at least 95% of the time. For the Longtail data set, 16 of the 142 feeds never returned usable data and were discarded. Of the remaining feeds, 87% were available at least 99% of the time, and 97% were available at least 95% of the time.

This data suggests that most feeds have high availability, but our sample period was too short to draw a strong conclusion. The presence of outdated URLs in the sample sets highlights the importance of a mechanism for occasionally updating the policies once they have been deployed. For example, a new policy could be delivered with other security updates to the application.

Entry Lifetime and Frequency.

Entry frequency—the rate at which new entries appear or old entries are altered—is a crucial parameter for feeds used with our system, because it determines how often fresh entries will be available. Among sites in the Popular data set where any new entries appeared during our survey period, the average entry frequency was 16.5 per day. Frequencies ranged from as high as 125 entries per day to only a single entry in our week long sample. For active sites in the Longtail data set, the entry frequency averaged a much lower 4.77 per day, but ranged as high as 65.8 per day.

Another important parameter, entry lifetime, measures the time from when an entry first appears in the feed until the time when it is updated, deleted, or displaced by newer content. The average entry lifetime for a source determines how long we can expect to be able to verify the source's content. We measured lifetimes for entries that died during our data collection period. If an entry was posted before we began collecting data, we used the posting date listed in the feed as the start of the entry's lifetime. We recorded the lifetime of 11779 entries from 102 sites in the Popular dataset. The average lifetime of an entry was 38.5 hours, but the averages for individual sites ranged from 3.7 hours to 43 days (among sites for which we had a statistically significant number of data points). From Longtail sources, we recorded 2404 entry lifetimes from 72 distinct sites. The average lifetime was 118 hours, and the averages for individual sites ranged from 6.9 hours to 90 days (among sites for which we had sufficient data).

RSS feeds typically include a fixed number of entries, with older entries aging out as newer ones are posted. The number of entries present in an RSS feed limits how many content chunks we can extract from it. We recorded the average number of entries present in each available feed. For sources in the Longtail data set, the average number of entries was 16.3; for sources in the Popular data set, the average was 22.3. Since feed sizes are usually constant, we expect to see an inverse relationship between the frequency of new entries and the average time entries remained in a feed, as shown in Figure 2. We observe that points in the horizontal tail of the distribution tend to come from the Popular data set, indicating that Popular sources are more likely to post several entries a day, leading to a short lifetime for each entry. Points in the vertical tail tend to come from the Longtail data set, indicating that those sites are more likely to post entries infrequency, leading their entries to have a long lifetime.

Policy Satisfaction.

To test how well the RSS feeds in our data sets would serve as sources for harvested challenges, we modeled a number of simple policies and verifiability requirements and calculated the percentage of the sample period when the policies and requirements would be satisfied.

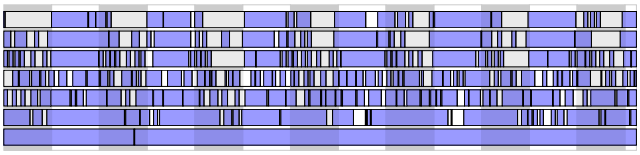
Generally, the more time that elapses between derivation and verification, the less precisely we need to know the freshness of the challenge. If a week has passed since the challenge was generated, it is less likely to matter whether the challenge was fresh a few hours earlier or later. This mirrors the inverse relationship between entry lifetime and frequency depicted in Figure 2. Thus, we expect RSS feeds to meet the needs of many kinds of applications.

In the first policy we modeled, which we label "Short," a deriver collects all entries from the source that are less than one hour old, and verifiers require at least one entry in the derivation to match the contents of the feed. We calculated satisfaction using this policy and two robustness standards, which required the policy to be verifiable a minimum of 6 hours and 12 hours after derivation. We tested these models with the Popular sources. Series A and B of Figure 3 show the percentage of sources that satisfied these requirements at least a given fraction of the time. In all, 13% of sources satisfied the 6 hour requirement at least 50% of the time, as did 7% the 12 hour requirement.

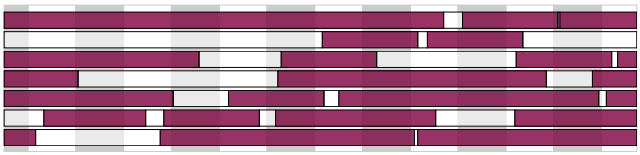
In the second policy we modeled, which we label "Long," derivers collect all entries from the source that are less than one day old, and verifiers require at least one entry in the derivation to match the contents of the feed. We calculated satisfaction using this policy and two robustness standards, which required the policy to be verifiable a minimum of 7 days and 14 days after derivation. We tested these models with the Longtail sources. Series C and D of Figure 3 show the percentage of sources that satisfied these requirements at least a given fraction of the time. In all, 24% of sources satisfied the 7 day requirement at least 50% of the time, as did 7% the 14 day requirement.

As these data show, not all sources are equally suitable for harvesting challenges; however, a significant fraction of sources were able to satisfy our model requirements at least half the time. Policy creators can achieve high robustness by combining data from a number of such sources. If satisfaction times were uniformly distributed and uncorrelated between sources, we would need to combine 10 sources with a suboptimal 50% satisfaction rate to reduce expected downtime to less than 90 seconds a day.

In practice, satisfaction times *were* correlated and not distributed uniformly, but policy creators will tend to select sources that are well suited for their policies. Here we depict the satisfaction of 7 selected sources modeling the "Short" policy with a 6 hour verifiability requirement. Each horizontal bar represents one source over the course of the week; shaded areas indicate times when challenges could be harvested and meet these requirements. At least one of the sources satisfied the policy at all times during the period; at least two sources satisfied the policy more than 90% of the time:



The next figure depicts satisfaction of 7 selected sources modeling the "Long" policy with a 7 day verifiability requirement. At least two of the sources satisfied the policy at all times during the period:



Our models indicate that RSS feeds may not be well suited to applications that require freshness guarantees within a time frame much shorter than an hour, ones that require verifiability over periods greater than a few weeks, or ones that require precise timestamps that are verifiable over long periods. Other sources may be more appropriate for such applications. For instance, realtime seismographic data [16] could provide very fine grained freshness (though from relatively few data providers), or historical stock quotes could provide coarse-grained timestamps that are verifiable many years into the future.

Other Considerations.

To ensure freshness of harvested challenges, a policy needs to prevent an attacker from predicting the future contents of sufficiently many of its sources. The predictability of an RSS feed depends heavily on the type of content being served. Content like wire service news headlines that are widely carried by different sites may be predictable minutes or hours before appearing on a particular source. Some sites, like the popular link aggregator Digg, allow members to influence the content that appears in their feeds by voting on candidate entries. An attacker could monitor changes in popularity during the voting process to predict future feed contents. Policy creators should avoid such feeds.

Posting times for many sources in our data sets were strongly correlated with local daylight hours for the site. This effect is clearly visible in first figure above, where the large vertical shaded regions indicate times from 7 p.m. to 7 a.m., PST. If freshness on the order of hours is required, policy creators might select sources from around the world, such major newspapers from each timezone.

We normally assume that an attacking deriver cannot sit between the verifier and the Internet sources and modify the verifier's view of their contents. Even if this is not the case, our model can remain secure in instances when the deriver is unable to corrupt enough sources to fool the verifier. The remaining danger can be mitigated by selecting feeds that are served using HTTPS.

7. RELATED WORK

Our work on harvesting challenges from the Internet touches on prior research in a number of areas. In this section we describe relevant related work and provide a context for our contributions.

Deriving Randomness.

The idea of extracting bits from a random source has been around for many years. Several works have shown how to extract randomness suitable for cryptographic applications from non-uniform sources of entropy, such as physical events on a local machine (see, for example, [8] and the references therein). Our problem is of a somewhat different flavor from this work. We want Alice not only to be able to extract sufficient randomness for her own use, but to be able to convince Bob that she derived it properly and freshly. Our primary challenges arise because the oblivious Internet sources that we wish to use are unreliable, which means the deriver and the verifier may not have the same view of these entropy sources. Our work focuses on the practical problems arising from this setting; we simplify the theoretical aspect of deriving uniform randomness by modeling our hash function as a random oracle [4].

Proofs of Work and Client Puzzles.

Dwork and Naor [10] first proposed the idea of using proofs of computational puzzles for mitigating spam. Adam Back independently proposed and implemented a similar system known as Hashcash [2]. Both the Dwork-Naor and the Back systems fail to prevent the pre-computation of puzzles by an attacker. The attacker can therefore begin his computation arbitrarily long before an attack is launched.

Juels and Brainard [12] observed that allowing arbitrary precomputation was problematic for protecting against DoS attacks, where an adversary might build up a collection of puzzle solutions over a long period of time and use them over a brief period to flood a server. Juels and Brainard proposed a solution they called "client puzzles" where a challenger would first send a fresh random challenge to a machine requesting a resource, and demand that the machine do a proof of work relative to the fresh challenge. Since an attacker does not know the challenge ahead of time, he is forced to perform all his computation on-line. Several other papers subsequently proposed other types of client puzzle solutions [1, 7, 19, 20].

One issue that arises from these systems is that a challenger must issue a random challenge. Unfortunately, this is not possible for non-interactive applications such as email. By leveraging our tool, we can provide suitable fresh challenges in many of these settings. Waters *et al.* [20] provide another motivation for our approach. In their system a client spends a somewhat larger amount of time solving a puzzle that he can use as a proof of work for many different servers. Since each server does not have the opportunity to challenge the client individually, the system requires a common trusted source for random challenges. In their work, Waters *et al.* suppose the existence of dedicated "bastion" servers for this purpose. By adapting their system to utilize our tool, we can potentially eliminate the need for such servers.

Borisov [5] examined the problem of deriving randomness from a community of peer-to-peer servers. Our approach is quite different in that we harvest challenges from oblivious Internet content providers and require less complex interaction to synthesize our challenges. In addition, the work of Borisov was initially inspired by our preliminary research (as noted in the related work section of [5]).

Client puzzle solutions must be carefully designed if they are to successfully mitigate attacks. Issues include the cost of verifying puzzles, the discrepancy between the computa-

tional resources of portable devices and standard processors, and the possibility that attackers will have control of large bot-nets. Many of the works cited above address these issues in particular settings. We stress that any system that uses our tool must carefully consider issues such as these in the context of the application it is protecting.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we addressed the problem of harvesting challenges from oblivious online servers. This setting presented us with a challenging set of issues in that we not only had to consider the usual issues of security and robustness in our application, but to deal with the unique problem that our Internet sources are unaware of their role in our system.

We addressed this problem by creating a framework with which a party can harvest a challenge from several sources and (non-interactively) convince another party that it was formed correctly. Our framework allows an application designer to specify a flexible policy that can be tailored to specific needs. We identified multiple security contexts where our tool may be valuable, including remote storage auditing and P2P Sybil attack prevention.

We implemented our methods in a software tool named Combine. Combine is able to use RSS feeds, historical stock quotes, and explicit randomness servers as sources for harvesting random challenges. We provided experimental data supporting our framework's practicality, and we built a proof-of-concept application, Postmark, that uses Combine to create an improved client puzzle system for email.

In the near future we plan to apply these techniques to build auditing mechanisms for existing systems. We will start by constructing an auditing component for a remote storage service. We expect that this process will teach us more about the subtleties of using harvested challenges in a systems environment. From a broader perspective, we will continue to search for additional applications where harvested challenges can be used to verify claims of distributed systems.

Acknowledgments

We would like to thank Dan Boneh, Ed Felten, Pat Lincoln, Chris Peikert, Amit Sahai, Shabsi Walfish, and our anonymous reviewers for useful comments and suggestions.

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

9. REFERENCES

[1] Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. DOS-resistant authentication with client puzzles. In *Security Protocols Workshop*, pages 170–177, 2000.

[2] Adam Back. Hashcash – a denial of service counter-measure. http://www.hashcash.org/hashcash.pdf, 2002.

[3] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In *CRYPTO*, pages 431–448, 1999.

[4] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[5] Nikita Borisov. Computational puzzles as sybil defenses. In *Peer-to-Peer Computing*, pages 171–176, 2006.

[6] Giovanni Di Crescenzo, Richard J. Lipton, and Shabsi Walfish. Perfectly secure password protocols in the bounded retrieval model. In *TCC*, pages 225–244, 2006.

[7] Drew Dean and Adam Stubblefield. Using client puzzles to protect TLS. In *10th Usenix Security Symposium*, pages 1–8, 2001.

[8] Yevgeniy Dodis, Ariel Elbaz, Roberto Oliveira, and Ran Raz. Improved randomness extraction from two independent sources. In *APPROX-RANDOM*, pages 334–344, 2004.

[9] John R. Douceur. The sybil attack. In *IPTPS*, pages 251–260, 2002.

[10] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, pages 139–147, 1992.

[11] J. Galvin, S. Murphy, S. Crocker, and N. Freed. Security Multiparts for MIME: Multipart/Signed and Multipart/Encrypted. RFC 1847 (Proposed Standard), October 1995.

[12] Ari Juels and John G. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *NDSS*, 1999.

[13] Martijn Koster. A standard for robot exclusion. http://www.robotstxt.org/wc/norobots.html, 1994.

[14] R. Kotla, M. Dahlin, and L. Alvisi. Safestore: A durable and practical storage system. In *USENIX Annual Technical Conference*, 2007.

[15] National Solar Observatory/Sacramento Peak. Images and current data. http://nsosp.nso.edu/data/.

[16] USGS Earthquake Hazards Program. Latest earthquakes in the world - past 7 days. http://earthquake.usgs.gov/eqcenter/recenteqsww/Quakes/quakes_all.php.

[17] RSS 2.0 specification. http://blogs.law.harvard.edu/tech/rss, 2003.

[18] Technorati: About us. http://www.technorati.com/about/, 2007.

[19] XiaoFeng Wang and Michael K. Reiter. Defending against denial-of-service attacks with puzzle auction. In *IEEE Symposium on Security and Privacy*, pages 78–92, 2003.

[20] Brent Waters, Ari Juels, J. Alex Halderman, and Edward W. Felten. New client puzzle outsourcing techniques for DoS resistance. In *ACM Conference on Computer and Communications Security*, pages 246–256, 2004.