EVE: Verifying Correct Execution of Cloud-Hosted Web Applications

Suman Jana
The University of Texas at Austin

Vitaly Shmatikov
The University of Texas at Austin

Abstract

We present a new approach to verifying that a completely untrusted, platform-as-a-service cloud is correctly executing an outsourced Web application.

1 Introduction

Web applications such as blogs, wikis, and online social networks are increasingly hosted on third-party, "platform-as-a-service" (PaaS) clouds. The owner outsources the execution of his application to a cloud provider—for example, Microsoft's Azure or Google's App Engine—and is periodically billed for the application's consumption of CPU time and network bandwidth. In a PaaS environment, the owner does not have much control or visibility into the provider's server infrastructure, virtual machines, data-center and network architecture, *etc.* The cloud is completely untrusted.

We describe a new approach to (1) verifying that cloud-hosted, interactive Web applications (*webapps*) are executed correctly, and (2) measuring the frequency and severity of violations without any trusted components in the cloud.

Our approach may be useful in several scenarios. First, the owner may want to determine whether his webapp is executed correctly by a particular PaaS cloud. Faults in the execution of a cloud-hosted webapp can result from bugs and/or misconfiguration anywhere in the provider's stack: network, OS, Web server, database back-end, *etc.* A malicious provider may try to reduce its expenses by executing the webapp incorrectly or incompletely. Our approach is agnostic to the source of potential faults and enables the webapp's owner to detect the consequences of faults in the opaque layers of the provider's infrastructure without direct access to these layers.

Second, the webapp's owner can use our approach to measure how the number of violations increases with the number of users and/or requests and thus estimate the scalability of his webapp in a given PaaS cloud.

Third, our approach can be used to compare the quality of service from different cloud providers by measuring the number of respective violations. Cloud-hosted webapps often rely on storage services supplied by the cloud provider. To achieve high scalability, these storage layers typically support only eventual consistency, a very weak form of consistency without any bounds on the time it may take for the storage to reach a consistent state. As a consequence, webapps relying on cloud storage may occasionally provide inconsistent responses to users' requests. For webapps of the type considered in this paper, the frequency and severity of consistency violations are key metrics for estimating the quality of a cloud provider's service.

The problem of verifying the execution of cloud-hosted webapps is different from the problem of verifying outsourced computation [3, 4]. The latter deals with non-interactive computation, where a single party supplies the computation and its inputs and verifies that the cloud provider performed the computation correctly. By contrast, webapps such as blogs and social networks are fundamentally interactive and involve an exchange of information with many users other than the webapp's owner. In a PaaS environment, the owner does not have access to the virtual machine(s) hosting his webapp and cannot directly observe its interactions with other users.

Key ideas. EVE, our prototype system for verifying the execution of cloud-hosted webapps, is based on two insights. First, we exploit the observation that many common webapps consist of a stateless front-end running on a Web server and a storage back-end (implemented as a database or key-value store) for keeping the persistent state. In webapps of this type, the front-end simply converts HTTP requests into reads and writes to/from the back-end and presents the results as HTML content. All webapp operations can thus be represented in terms of *reads and writes to an object store*, with or without access control. For a large class of webapps, this property

allows us to reduce the problem of verifying whether an application is executing correctly to the problem of verifying whether an object store is consistent.

Second, multi-user webapps require *collaborative verification*. EVE depends on a few cooperating users called "witnesses" who keep logs of their interaction with the cloud-hosted webapp. Under standard fault models (*e.g.*, random faults) and as long as witnesses are indistinguishable by the cloud provider from regular users, deviations from the application's expected behavior will be detected with high probability.

In EVE, the cloud is completely untrusted. The key distinction from other systems for verifiable or accountable cloud computation is that EVE aims to verify the externally visible I/O behavior of the cloud-hosted webapp, as opposed to the actual computation performed inside the cloud. Verifying the latter requires a source of trust that is not controlled by the cloud provider (for example, a trusted hypervisor or hardware module). PaaS providers today do not support this model. Furthermore, enforcement based on trusted cloud components ends at the virtual-machine boundary and does not detect network faults after packets have left the virtual machine. By contrast, EVE measures the "black-box" behavior of the cloud-hosted webapp from the end user's point of view and is thus deployable on existing PaaS platforms.

2 Related Work

GridCop [8] uses compiler-based techniques and periodic beacon exchanges to check the progress and correctness of a remotely executing program. It trusts the cloud provider's infrastructure such as the JVM and OS. By contrast, EVE is designed to verify Web applications executing in a completely untrusted cloud.

Several proposals [2, 6] rely on trusted hardware ("Trusted Platform Module" or TPM) to detect modifications in remotely executing programs. Existing PaaS platforms such as Google's App Engine do not provide access to their internal TPMs, thus TPM-based techniques cannot be used to verify applications whose execution has been outsourced to one of these platforms.

Pioneer [7] uses fine-grained delay variations to detect modifications in programs running on a remote host. This technique only works in smaller networks where network delays are negligible and does not appear to be applicable to PaaS scenarios.

Accountable virtual machines (AVMs) [5] ensure correct execution of remote processes. AVMs rely on the server to record all incoming and outgoing messages. Existing PaaS platforms do not provide this functionality to their clients. By contrast, EVE does not rely on any trusted components inside the cloud, nor assumes anything about the virtual machine in which the application is executed. Because EVE operates at a higher level and has better visibility into the application's semantics, it can record only those application-specific operations that are important for verification.

3 Verification Methodology

3.1 Object-store semantics of webapps

EVE represents the semantics of webapp operations in terms of reads and writes to an abstract "object store" whose interface is similar to a standard key-value store. We found that this mapping from application-level operations to one or more object-store operations is straightforward for many common webapps. In our WordPress case study, we constructed the mappings manually (see Section 4.2), but they can also be generated automatically. This is a topic for future work.

In the object store, objects are identified by unique ids and have *properties* whose values are either arrays, or single values. By default, any property that has not been written to has value NULL. The result of an operation on an object is success or failure.

Each write specifies the values for all properties of the target object. Partial writes are modeled by reading all of the object's property values, modifying some of them, and writing all property values back into the object. A "create" operation is simply a write with the desired value for the creation-time property. A "delete" operation is a write that sets the values of all properties to NULL. We assume that the webapp keeps track of the identity of the user (userid or IP address) who performed the last write by storing it in a special property of the object and returns this identity with every read.

3.2 Client logs

Each client participating in verification as a "witness" (see Section 3.3) maintains a log of its interactions with the cloud-hosted webapp. In the log, every webapp operation is represented in terms of object-store operations. Of course, the client observes only HTTP requests and responses, not the actual object-store operations performed in the cloud, but the mappings described in Section 3.1 are used to convert the former into the latter.

Each entry in the log contains the following: (1) begin timestamp, (2) end timestamp, (3) operation type (read or write), (4) result (success or failure), (5) object id, (6) values of the object's properties, including the identity of the user who performed the last write of this object. All timestamps are generated by the client. We assume that clients' clocks are loosely synchronized. The correctness guarantee provided by EVE's verification pro-

cedure decreases gradually as the skew between clients' clocks increases.

Logging is automated by instrumenting the webapp's client-side code. Once a user agrees to become a witness, his logs are automatically and transparently sent to the verifier running on a separate server maintained by the webapp's owner or one of the clients. The cloud provider cannot tamper with the logs: either they are sent over a network connection that does not traverse the provider (*e.g.*, email or peer-to-peer), or their integrity is cryptographically protected by an HMAC or digital signature. To avoid key distribution, the current prototype of EVE relies on the former mechanism.

3.3 Collaboration

Collaborative verification of multi-user webapps in EVE depends on a subset of users (*witnesses*) who keep logs of their interaction with the webapp and periodically send them to the *verifier*, as described in Section 3.2. The verifier checks whether the logs are *consistent* with the intended operation of the webapp using the efficient streaming algorithm described in Section 3.4.

Witnesses can be recruited using incentives—for example, free use of the webapp. They behave like normal users in all other respects. It is important that the cloud provider not be able to distinguish witnesses from regular users; otherwise, a malicious provider may avoid detection by executing the webapp incorrectly only for regular users. If faults occur randomly and independently for each user with probability p_e and the fraction of indistinguishable witnesses among users is p_w, then the probability that a faulty cloud provider successfully avoids detection after n interactions is $(1-p_e p_w)^n$.

3.4 Streaming verification of consistency

There are different types of consistency. *Atomicity* requires any read of an object to return the values of the latest write to that object. With multi-user webapps, however, clients may not know the order in which the webapp processes concurrent writes (for example, simultaneous comments to a popular blog post). It is not possible to verify atomicity without an online central proxy mediating all interactions between the clients and the webapp, which is unrealistic for cloud-hosting scenarios.

Instead, EVE verifies a weaker property. *Regularity* requires any read of an object to return the value of either the last write, or one of the concurrent writes. Our consistency verification algorithm is similar to the algorithm of Anderson et al. [1]. As in [1], we define three possible relations between any two log entries A and B referring to the same object:

Algorithm 1 Streaming verification of consistency

```
Collect client logs and add to existing log
Create object-specific logs based on objectid
for each object-specific log do
  Sort the log based on begin timestamp
  beginoffset = 0
  Find a READ R_sep where offset(R_sep) > min_increment_size and ((R_sep.beginTime) − current_time) > max_delay
  while no such read R_sep found do
    sleep(t) and restart
  end while
  endoffset = offset(R_sep)
  for all operations O such that O||R_sep do
    endoffset = endoffset + 1
  end for
  Verify consistency of partial object-specific log(0, endoffset) using Algorithm 2
  if partial object-specific log inconsistent then
    return INCONSISTENT
  else
    for each operation O in partial object-specific log(0, endoffset) do
      if O < R_sep and O is not dictating write of R_sep then
        Remove O from log
      end if
    end for
  end if
end for
```

$A < B$ if A.endTime < B.beginTime
$A > B$ if A.beginTime > B.endTime
$A||B$ if (A.beginTime < B.endTime < A.endTime) or (B.beginTime < A.endTime < B.endTime)

The algorithm of Anderson et al. is a relatively expensive offline algorithm and does not scale with the number of log entries. By contrast, our Algorithm 1 is a streaming algorithm which is orders of magnitude faster for large logs than the algorithm of Anderson et al. Instead of verifying all log entries at once, our algorithm divides the log into chunks and verifies each chunk separately. To preserve correctness, some of the entries in a chunk must overlap with those of the immediately following chunk. Once a chunk is verified, its non-overlapping entries can be removed from the log.

Our Algorithm 2 for verifying consistency of partial object-specific logs by building precedence graphs is similar to the corresponding algorithm of Anderson et al. In collaborative verification, the verifier receives logs only from collaborating witnesses. Verification is thus partial and provides probabilistic guarantees (see Section 3.3). Algorithm 2 only considers reads R where the

Algorithm 2 Verification of consistency of partial object-specific logs

```
for all operations A and B such that A < B do
    add an edge A → B
end for
for every read R in input log do
    Find R's dictating write W such that
    (W.property_values==R.property_values)
    if no dictating write W then
        return INCONSISTENT
    else if not (W||R) then
        Add an edge W → R
    end if
end for
for every write W' and read R, where W' < R do
    Add an edge W' → W, where W is R's dictating
    write
end for
if any cycle in resulting graph then
    return INCONSISTENT
else
    return CONSISTENT
end if
```

user who performed the latest write on the object belongs to the witness set (recall that the identity of this user is stored in a special property of the object and thus returned with every read). If both witnesses and non-witnesses perform writes on the same object, the verification algorithm effectively ignores the consequences of non-witness writes because it cannot check their correctness. This does not break the overall probabilistic guarantees of EVE. Because the cloud provider cannot tell the difference between witnesses and non-witnesses, faults cannot affect only non-witnesses.

In practice, webapps can suffer from temporary consistency violations even during benign execution because many popular back-ends are eventually-consistent key-value stores which may occasionally provide inconsistent results. Following Anderson et al., our algorithm, too, not only detects consistency violations, but also approximately measures the extent of violations over a period of time, thus enabling the webapp's owner to estimate the quality of the cloud provider's service.

4 Case Study: WordPress

To illustrate the capabilities of EVE, we apply it to WordPress, a popular blogging Web application. The front-end of WordPress is a set of PHP scripts running on a Web server, the back-end is a MySQL database. WordPress lets users publish blog posts and add comments to other users' posts. In our prototype, we only verify a subset of the operations supported by full WordPress. Because WordPress has a complex role-based access-control model, we leave verification of access-control enforcement in cloud-hosted WordPress to future work.

4.1 Objects in WordPress

Our abstract "object store" model for WordPress involves two types of objects: posts and comments. A *post* object is a user-created blog post. It has six properties: postid, content, authorname, isdraft, and timestamp. "postid" is a unique identifier, "content" contains the content of the post, "timestamp" contains the time when the post was created. WordPress allows users to store unpublished posts in the server as drafts. The "isdraft" property is set to 1 if the post object is an unpublished draft.

A *comment* object has eight properties: commentid, content, authorname, authorwebsite, timestamp, parent_postid, parent_commentid, and depth. Similar to post objects, "commentid" is a unique identifier, while "content" contains the content of the comment. WordPress associates each comment with the corresponding blog post, whose postid identifier is stored in the "parent_postid" property of the comment object. Comments in WordPress can be nested to arbitrary depth, *i.e.*, it possible to add a comment to a comment. The "parent_commentid" property stores the commentid of the parent comment object, while "depth" indicates the number of levels from the top-level comment. For example, a comment added directly to a post has depth 0, while a comment added to a top-level comment has depth 1.

4.2 Mapping WordPress operations to object-store operations

The key idea behind our approach is to map application operations to object-store operations. Table 1 shows the manually constructed mapping for common WordPress operations. WordPress supports creation, modification, and deletion of both posts and comments. Whenever a user creates a new post or comment, WordPress returns a unique identifier. The space of WordPress identifiers for posts (*id*) overlaps with the space of identifiers for comments (*commentid*). To make identifiers unique, we prepend object type (0 for posts, 1 for comments) to the WordPress-generated identifier and use the resulting value as the object-store identifier (*osid*).

4.3 Experimental results

We set up a WordPress server and multiple clients who interact with the server using XML-RPC. The server was initialized with 8 blog posts. The clients randomly try to read these posts or write comments to them and log

Table 1: Mapping WordPress operations to object-store operations

WordPress operations	Object-store operations
readPost(user,wpid)	readObj(user,osid)
id=createPost(user,content, authorname,tags)	writeObj(user,osid,content,authorname,...)
modifyPost(user,id,content)	readObj(user,osid); writeObj(user,osid,content,authorname,...)
deletePost(user,id,content)	writeObj(user,osid,NULL,NULL,...)
id=createDraft(user,content, authorname,tags)	writeObj(user,osid,content,authorname,isdraft=1)
modifyDraft(user,id,content)	readObj(user,osid); writeObj(user,osid,content,authorname,isdraft=1)
createPostFromDraft(user,id)	readObj(user,osid); writeObj(user,osid,content,authorname,isdraft=0)
readComment(user,commentid)	readObj(user,osid)
commentid = createComment(user, content,authorname,...)	writeObj(user,osid,...)
deleteComment(user,commentid)	writeObj(user,osid,NULL,NULL,...)

their interactions with the server. The resulting joint log is submitted to EVE, which checks it for consistency.

To compare the performance of our consistency verification algorithm with the algorithm of Anderson et al. [1], we implemented the latter as well. To keep the comparison fair, in both cases the entire log is loaded into memory and entries sorted before processing, although EVE can first sort the log and then process it incrementally for better performance. For this test, *min_increment_size* in Algorithm 1 is set to 100. If an R_{sep} is not found, then, instead of sleeping, Algorithm 1 is modified to process the rest of the entries together. All tests were done on a laptop with an Intel Core Duo 2.00 GHz CPU and 2 GB RAM. Results are shown in Fig. 1, demonstrating better scalability of our algorithm. When the number of log entries exceeded 1600, the algorithm of Anderson et al. ran out of memory.

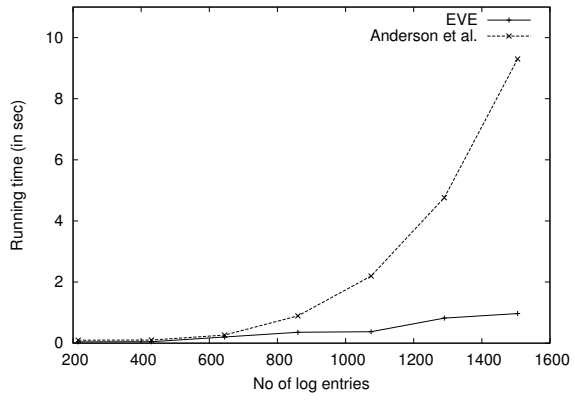

Figure 1: Performance comparison of EVE and Anderson et al.'s algorithm.

5 Future Work

5.1 Generating operation mappings

EVE is based on the observation that many Web applications can be "deconstructed" into a stateless front-end, which is responsible for processing clients' requests and generating HTML responses, and an object-store back-end, which is responsible for keeping the entire state of the application. For such webapps, application-level operations can be represented in terms of reads and writes to the underlying object store. The problem of verifying whether the application is executed correctly in a PaaS cloud can then be reduced to the easier problem of verifying whether distributed reads and writes are correctly reflected in a cloud-based object store.

One of the challenges for future development of EVE is to automate this mapping from application-level operations such as clients' HTTP requests to object-store operations. One option is to generate the mappings as part of the application development process. Another option is to add an extra layer of indirection by implementing a "wrapper" library for each actual store. This library would (1) perform the object-store operation by submitting a query to a concrete SQL database or a key-value store, and (2) generate an abstract representation of this operation in a form suitable for verification using the algorithms of Section 3.4. When deployed on a PaaS cloud, the webapp would use an appropriate library on top of a concrete object store as its storage back-end.

When the verifier receives a log of application-level operations from "witness" clients, he replays these operations to a special local version of the webapp whose storage library has the same interface as described above but whose implementation only contains (2). In other words, it logs what the effect of an operation would have

been had it been executed over an actual object store. The resulting logs are then checked for consistency.

In addition to automatically converting application operations into object-store operations for a large class of webapps, this approach also has the benefit of improving portability of webapps. If a webapp uses an abstract object-store library interface as its back-end rather than a concrete store, it can be transparently moved from one object-store implementation to another without any changes to the application code.

5.2 Dealing with malicious witnesses

The collaborative verification approach described in Section 3.3 assumes that witnesses who submit logs for verification are honest. A malicious witness can potentially send incorrect logs, producing false negatives (tell the webapp's owner that the webapp is being executed correctly even though the execution is faulty) or false positives (tell the owner that the execution is faulty even though it is correct).

We are not concerned about false negatives because a malicious witness only hurts himself if he receives incorrect outputs from the cloud but reports that everything is Ok. Note that the cloud provider cannot collude with malicious witnesses to hide all faults unless faults are limited to malicious witnesses (in this case, the damage from incorrect execution does not affect honest users). False fault reports, on the other hand, may unfairly "frame" the cloud provider.

One possible approach to dealing with untrustworthy witnesses is to have the verifier record the discrepancies between the reports of individual witnesses and those of the majority of witnesses. This helps detect witnesses who consistently report faults while the majority observes only correct execution. Developing a robust reputation scheme for this setting is a topic for future research.

An alternative is cryptographically enforced *accountability*. If the server cryptographically signs all messages and the client submits them with his log, the verifier can check if the log is correct. This requires online cryptography for every webapp operation.

5.3 Verifying access-control semantics

Some webapps involve access control (for example, only the author of a blog post should be able to see it until the post is published). For such webapps, it is necessary to verify that the interaction between the witness and the webapp satisfies the application's access-control semantics. Access-control information is stored in the back-end just like any other persistent data and retrieved before allowing or denying user-requested operations. Without loss of generality, we assume that access-control information is kept in special access-control objects. Whenever a new user is added or privileges of an existing user changed, the webapp writes to the corresponding access-control object.

Reads from access-control objects are not performed explicitly by users. Instead, any read or write on any user-created object (including access-control objects) also generates a read from the corresponding access-control object of that user. These reads can then be checked for consistency to ensure that access control is being enforced correctly.

Acknowledgments. We are grateful to Mihai Christodorescu for discussing many of the ideas and collaborating on early versions of this work. This research was partially supported by the NSF grants CNS-0746888 and CNS-0905602 and the MURI program under AFOSR Grant No. FA9550-08-1-0352.

References

[1] E. Anderson, X. Li, M. Shah, J. Tucek, and J. Wylie. What consistency does your key-value store actually provide? In *HotDep*, 2010.

[2] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP*, 2003.

[3] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, 2010.

[4] P. Golle and I. Mironov. Uncheatable distributed computations. In *CT-RSA*, 2001.

[5] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *OSDI*, 2010.

[6] D. Levin, J. Douceur, J. Lorch, and T. Moscibroda. TrInc: small trusted hardware for large distributed systems. In *NSDI*, 2009.

[7] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP*, 2005.

[8] S. Yang, A. Butt, Y. Hu, and S. Midkiff. Trust but verify: monitoring remotely executing programs for progress and correctness. In *PPOPP*, 2005.