# Language-based isolation for cloud computing: An analysis of Google App Engine

Nicholas Carlini UC Berkeley
Aaron Wong UC Berkeley
William Robertson UC Berkeley
David Wagner UC Berkeley

**Abstract**

We study the security of language-based isolation, one way that cloud computing providers can isolate client applications from each other. In particular, we analyze the security of Google App Engine for Java, which uses language-based isolation to protect applications, and use this as a case study in the strengths and weaknesses of language-based isolation. In this paper, we introduce a set of attacks that allow a malicious application to attack other applications hosted on the same JVM. If an attacker can get his or her application on the same JVM as a target, it is possible for the attacker to exploit shared channels, such as static globals and the intern pool, to both observe the target application and manipulate its state. These attacks do not appear to threaten Google App Engine, but shed light on the security of language-based isolation. We identify possible defenses for each attack we present. Our analysis suggests that language-based isolation can be effective for application protection, particularly if augmented with a carefully designed mechanism for assigning applications to hosts.

## 1 Introduction

Cloud providers have become a valuable enabling technology for deploying scalable applications to the web. Cloud providers are typically classified according to the level of abstraction they provide to application developers. At the lowest level, cloud infrastructure, or "Infrastructure as a Service" (IaaS), provides developers with virtual network, storage, and computing resources. Popular examples of this form of cloud computing include Amazon Elastic Compute Cloud (EC2) [1] and Rackspace Cloud [12].

Alternatively, "Platform as a Service" (PaaS) provides a higher level of abstraction. Examples of popular cloud platforms include Google App Engine (GAE) [5] and Microsoft Windows Azure [9]. In this model, the cloud provider exposes a software stack and API that application developers use to create applications. Such platforms generally leverage cloud infrastructure in order to scale deployed applications, and trade off flexibility and control over application design for greater ease of development.

There are some security risks associated with cloud computing. The multi-tenancy of applications within a single physical or virtual machine — potentially from mutually distrusting principals — opens the door to security threats from malicious applications that share resources with victim applications. Previous work has studied the potential for applications to exploit multi-tenancy in cloud providers that use hypervisors and virtual machines for application isolation [13]. That work found that a malicious application could identify where a target application would reside in the Amazon's cloud infrastructure and, furthermore, mount side-channel attacks against a target virtual machine in order to extract sensitive information.

Hypervisor-based isolation is not the only means by which cloud providers protect applications from each other. In particular, several providers use language-based techniques to isolate applications cohabiting a single virtual machine. One prominent example of this approach is Google App Engine [5]. GAE is a cloud platform that allows developers to deploy either Java or Python or Go-based applications that leverage Google's hosting and software stack. In the Java-based version of GAE, applications may share a virtual machine with applications from potentially malicious users, but a number of mechanisms put in place by Google attempt to prevent applications from interfering with one another (e.g., resource limits, application-specific classloaders, and class whitelists).[1]

In this work, we examine the effectiveness of language-based isolation for cloud platforms, using the Java-based version of GAE as a case study. Our study demonstrates that while language-based mechanisms do provide a degree of isolation between applications, this isolation is by no means complete. By identifying and exploiting a number of information channels shared between applications, we demonstrate the potential for attacks against confidential application data.

To summarize, in this paper, we make the following contributions:

- We identify a number of attacks against privacy and availability of applications hosted on cloud platforms that use language-based isolation.
- We evaluate the viability of these attacks in a real system, and evaluate their impact on a set of applications.
- We suggest potential steps to mitigate the impact of these attacks.

[1] For a detailed description of Google App Engine, please refer to §3.

The remainder of this paper is structured as follows. §2 outlines our threat model for cloud-based applications in a language-based isolation setting. §3 presents an overview of GAE. §4 discusses the attacks that we have identified against GAE. §5 evaluates the threat these attacks pose to applications hosted under GAE. §6 discusses the impact of these attacks and how they might be addressed. §7 presents related work. Finally, in §8 we conclude and present avenues for future work.

## 2 Threat Model

We focus on the potential for a malicious application to attack other applications. We assume the attacker is a client of Google App Engine and thus can submit a malicious application for execution. We consider two threat models:

- *Targeted attack:* The attacker has a particular target application in mind, and would like to observe or influence the target application's behavior.
- *Opportunistic attack:* The attacker is interested in attacking any application it can (e.g., any application it happens to be co-resident with).

We assume that multiple mutually distrusting applications can be co-resident in a single Java Virtual Machine (JVM), and thus share access to resources. This introduces the possibility that a malicious application may be able to attack other applications co-resident on the same JVM, by observing or modifying shared resources. We study the degree to which malicious applications can control where they are located in the provider's cloud infrastructure.

As in the standard web attacker model, a malicious application can communicate over the network with a victim application through the standard interfaces exposed by that victim, but can not intercept other network traffic. We assume that malicious applications cannot violate the memory-safety or type-safety guarantees provided by the Java language runtime. For instance, applications cannot directly inspect objects belonging to another application, and they cannot directly modify a victim application's code or data.

## 3 Background: Google App Engine

Google App Engine allows developers to write Java servlets and run them on Google servers. The applications do not run persistently: rather, when a request arrives, the application is started on one of Google's JVMs and it serves the request. The Google cluster consists of many servers, and each server runs multiple JVMs. There is a scheduler component which assigns each application to a JVM on a server. Applications may be periodically re-located to another JVM or another server.

GAE overrides and restricts access to several Java libraries, to sandbox GAE applications. GAE applications are not permitted to define native methods or write to the file system, and they are allowed to read from, but not write to, an application-specific subdirectory of the filesystem. Furthermore, they are not allowed to spawn new threads or create new thread groups.

Applications have no view of the internal local-area network and cannot directly open network connections. Instead, network requests are proxied through a URL fetch service, which allows outgoing connections on port 80 (HTTP) and 443 (HTTPS). Requests sent to any other port are blocked.

GAE imposes quotas on many aspects of execution. Applications must respond to requests in a limited timeframe. Hard limits are imposed on the rates which applications may send emails and HTTP requests, as well as the rate at which they make calls to various APIs. Other resources, including total CPU time, total data storage, and total incoming and outgoing bandwidth, are subject to a two-tier quota: a restrictive quota for free access, and a less restrictive quota for paid access.

## 4 Attacks

We envision that attacks will involve two basic steps:

1. *Become co-resident:* First, the malicious application must arrange to be run in the same JVM as the target application. The attacker must be able to detect when this has successfully occurred.
2. *Exploit shared state:* Once running on the same JVM as the target application, the malicious application must observe or influence the behavior of the target application, through some shared resource.

We analyze the ways that an attacker might achieve these goals below.

### 4.1 Becoming Co-resident

Google App Engine does not explicitly expose which server or JVM the application is executed on, nor does it intentionally provide applications with any control regarding which server or JVM they run on. However, we identify side channels that allow a malicious application to deduce a great deal of this information.

**Identifying the server.** GAE allows applications to invoke `System.nanoTime()`, which returns the number of nanoseconds since the server was booted. GAE applications can also access the current time, accurate to a millisecond. By subtracting these two numbers, an application can deduce the time at which the local server was last booted, accurate to within a millisecond. Assuming no two servers are booted in the same millisecond, this gives a way to "fingerprint" the server that the application is currently executing on. In particular, this lets an application detect when it has been assigned to a new server, and recognize when it has been assigned to a server it was previously running on.

**Identifying the JVM.** There are many ways for an application to "fingerprint" the JVM it is running on. One simple way is to use the identity hash code of any static global variable, interned string, or `Class` object. Because these hash codes are effectively pseudorandom, but constant throughout the lifetime of the JVM, they uniquely identify the JVM. Alternatively, an application may recover the seed of `Math.random()` (see Appendix B and Appendix C) and use this to identify the JVM. For instance, if the application is moved from JVM A to JVM B and then back to JVM A, these methods can be used to detect that fact.

**Shifting to a different JVM/server.** The GAE infrastructure does not intentionally provide any mechanism for an application to specify which server or JVM it should run on. However, through our experimentation with GAE, we have discovered two ways that an application can indirectly control which JVM and server it is running on: by applying heavy load, or causing an `OutOfMemoryError`.

Applications on GAE run from a single JVM on a single server when under low load. When an application begins to receive many requests, the infrastructure will split off multiple instances of the application onto multiple JVMs, still on the same server. If load increases even further, exceeding the capacity of a single server, the application will be replicated onto at most two other servers. Therefore, at any point an application may be running on up to three servers. Each of these servers allows up to ten JVMs, setting an upper limit of 30 JVMs per application.

This provides one way for an attacker to have a limited degree of influence on the servers that his malicious application executes on: the attacker can send many requests to his own application, raising its load until it is replicated on to 30 JVMs, and hope that the target application is running on one of these. However, this requires a significant amount of network traffic, and the attacker's ability to influence where his application runs is sharply limited.

We also discovered a more efficient way to switch JVMs: if the application triggers an `OutOfMemoryError`, it will be moved to another JVM on the same server.[2] The application does not need to actually exhaust available memory; rather, it is sufficient to throw an `OutOfMemoryError`. Other errors simply crash the application and do not cause it to shift to a different JVM. We know of no way to force an application to move to a different server other than by applying a high load and waiting for it to be automatically replicated on another server. Also, we do not know of any way to cause the application to migrate outside of the set of three servers allocated to that application.

As a consequence, we are not aware of any realistic way to mount a targeted attack. A single malicious application can only reach a tiny fraction of the entire Google cluster. An attacker could plausibly register many malicious applications, perhaps using multiple false identities, but it would likely be very challenging to reach a large fraction of the Google servers. Opportunistic attacks may be a greater threat: if there are any sensitive applications running on one of the three servers that a malicious application can reach, and if the attacker can recognize this fact, the malicious application can arrange to be co-resident with it.

In the rest of this section, we explore the attacks that become possible if the malicious application manages to become assigned to the same JVM as some target application. The relevance of these attacks to Google App Engine is premised upon the assumption that it is possible for two distrusting GAE applications to be assigned to the same JVM. We emphasize that *we have not verified this critical assumption*.

### 4.2 String Interning

It is possible for two strings to have the same value but to not be identical. That is, given two strings `x` and `y`, `x.equals(y)` may return true, even though `x != y`. Java allows strings to be "interned." Interned strings that share the same values are guaranteed to be identical. Some applications use interning to reduce the cost of string comparisons. We found that interning introduces a side channel that allows an attacker to determine whether or not a particular string has been interned before.

**The specific timing attack.** The JVM maintains a pool of strings that have been interned. When a string is interned, either the string already present in the pool is returned, or a new string is created, added to the pool, and returned.

Internally, the intern pool is a hash table with linked-list chaining. When an application calls the `intern()` method, it first computes the hashcode and looks up the corresponding linked list. The `intern()` method then does a linear scan through the linked list to check if any of these strings has the same value as the one being interned. If a match is found, it is returned. If not, a new string is created and added to the front of the linked list, and the new string is returned.[3]

[2] When the application is moved to a new JVM, the old JVM is not shut down. We verified this by running an application which sets static globals to a random value and throws an `OutOfMemoryError`, repeating this process until, eventually, the application returns to a JVM where the static globals have the same values. We observed that the application visits many different JVMs, always on the same server. On average, the application returns to a previously seen JVM after switching JVMs approximately 120 times.

[3] Although the Java API does not require a new string to be returned, this occurs in practice. If the old string was returned instead, a timing attack would not be required.

This opens up a timing attack that can check whether a target string $s$ is present in the intern pool. The attack involves a preparation phase, a waiting phase, and an observation phase. To prepare, the attacker generates and interns thousands of different strings with the same hashcode, but not the same value, as $s$ (see Appendix D). The attacker then waits; during this time, the target application might intern $s$, and the attacker's goal is to determine whether $s$ was interned during this time. Finally, in the observation phase, the attacker interns the string $s$ and times how long this takes. A short delay indicates $s$ has recently been interned, whereas a long delay indicates the `intern()` method searched through the entire linked list in its attempt to find $s$.[4]

**Information leakage through interning.** An application which interns sensitive data — directly or indirectly — is vulnerable to attacks that reveal this data. An attacker may also be able to determine the internal state of an application by observing which strings it interns. If it is known an application only interns specific strings upon entering a given state, the presence of those strings indicates whether the application has entered that state.

**Detecting application presence through interning.** An attacker could learn whether a target application is on the same JVM as the attacker by mounting an algorithmic denial-of-service attack [3]. Many GAE APIs—such as those that might be used by a target application—intern hundreds of strings, when invoked. A malicious application could intern thousands (or millions) of strings with the same hashcode as one or more of these strings. This will slow down the target application when it invokes the GAE API, if it is on the same JVM. The attacker can then make HTTP requests to the target application; if they take longer than usual, the target application is most likely on the same JVM as the attacker.

A more efficient attack allows an attacker to detect if it is co-resident with a particular target application by checking for certain strings in the intern pool. A typical application contains many application-specific string literals, which are interned automatically. Every application we have examined has literal strings not found in any other application. If the malicious application finds one of these strings in the intern pool, it is highly likely the target application is on the malicious application's JVM.

### 4.3 java.util.Random

We found an attack on Java's `Random` object that allows an attacker to determine how many `Random` objects have been created in an elapsed interval. This attack relies on the fact that the `Random` object is not cryptographically secure, and, given two consecutive outputs, the original seed can be deduced (see Appendices A, B).

**Determining number of Random objects created.** The default constructor for `Random` seeds it with the current time in nanoseconds plus a static counter, which is incremented on every call. The static counter is shared among all applications on the same JVM. An attacker can determine the value of the static counter at a given point in time by subtracting the time when a `Random` object was seeded from the actual seed of this object. Obtaining the time the object was seeded is a nontrivial task, however it is possible. By obtaining the value of the counter two times, an attacker can deduce the number of Random objects created in this timeframe.

### 4.4 Math.random()

An attacker who is on the same JVM as a target application can manipulate `Math.random()` to both learn which random numbers the target application has been receiving and to force the target application to receive specific random numbers. `Math.random()` relies on a static `Random` object, shared between applications on the same JVM. `Math.random()` invokes `nextDouble()` on that `Random` object.

**Determining the Math.random() seed.** Any application which relies on `Math.random()` to produce unpredictable pseudo-random numbers is vulnerable to attack. The `Random` object in Java is a linear congruential generator. When a malicious application is running on the same JVM as a target, the malicious application can request a random number from `Math.random()`, infer the seed of the underlying `Random` object (see Appendix B), and then predict the sequence of all random numbers that will be produced by `Math.random()`. In this way, if the target application uses `Math.random()` to generate random numbers, the malicious application learns what random numbers the target application receives.

**Manipulating the result of Math.random().** In fact, a malicious application can even influence the random numbers that a target application receives. The attacker cannot completely specify the random number returned to the target application, but if there is some criteria determining which numbers are acceptable, the attacker can ensure the target application will receive an acceptable number by repeatedly generating random numbers until the next output of `Math.random()` is acceptable. If a fraction $p$ of numbers are acceptable, the malicious application will need to invoke `Math.random()` about $\frac{1}{p}$ times. As an example, if the target application rolls a 6-sided die by computing `(int)(Math.random()*6)+1`, and if the attacker wants it to receive the die roll 5, then the attacker can repeatedly invoke `Math.random()` until the next number in the sequence is between $\frac{4}{6}$ and $\frac{5}{6}$.

[4] Because this process adds the string $s$ to the intern pool when it is not already present, this attack can only be performed once for each target string during the lifetime of the JVM.

| $n$ | String Pos | Mean (ns) | Median (ns) | $\sigma$ (ns) |
|---|---|---|---|---|
| 1000 | Start | 39454 | 29731 | 65996 |
| | Missing | 98375 | 72177 | 130464 |
| 2000 | Start | 39636 | 30210 | 50624 |
| | Missing | 130722 | 99305 | 141494 |
| 3000 | Start | 38582 | 30221 | 55223 |
| | Missing | 154249 | 124454 | 131433 |

Table 1: Time required to intern strings.

This attack relies on the attacker being able to determine when the target application will request the random number. This is possible under two situations: first, the target may call `Math.random()` at regular (or otherwise predictable) intervals; or second, we may know that a user of the target application is about to take an action that causes a call to `Math.random()`.

### 4.5 Shared Channels

Any state shared throughout the JVM is a potential channel through which information can be leaked. Globally accessible objects (e.g., those referenced by a public static variable) are a major source of shared channels. An application that either reads from or writes to a globally accessible, mutable object is potentially open to attack, since a malicious application on the same JVM can modify and observe the state of these objects.

There are many instances of mutable, globally accessible objects throughout the JVM, including both public static non-final fields as well as public static final fields that point to a mutable object.

## 5 Evaluation

We ran several experiments to evaluate the feasibility of the attacks described in the previous section.

### 5.1 String Interning

The string interning attacks require the attacker to distinguish between two cases based upon the time it takes to intern a string $s$: case 1: the intern bucket contains $n+1$ strings, and $s$ is at (or near) the front; case 2: the intern bucket contains $n$ strings, but not the string $s$. To verify that these cases can indeed be distinguished, we measured the distribution of these times empirically. In our experiments, we randomly generated $n$ strings with the same hashcode as $s$ and interned them, then either interned string $s$ or not, then timed how long it takes to intern $s$. We considered $n = 1000$, 2000, and 3000, for 100,000 trials each. Summary statistics are in Table 1.

The empirical distributions let us find the optimal procedure for distinguishing between these two cases. In particular, it suffices to select a threshold: if the time taken is above the threshold, we infer that $s$ was not previously interned, otherwise infer that it was. We calculate the threshold that minimizes the total error rate.

| $n$ | Threshold (ns) | Total error rate |
|---|---|---|
| 1000 | 50606 | 0.1050 |
| 2000 | 74384 | 0.0744 |
| 3000 | 97719 | 0.0533 |

Table 2: Accuracy rate at distinguishing whether a string is already present in the intern pool.

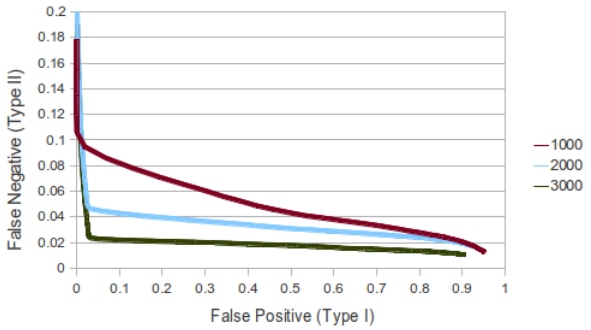


Figure 1: Error rates for different thresholds.

There are two possible types of errors: false positives (Type I errors), which occur when we predict the string was in the hash table when it actually was not, and false negatives (Type II errors), which occur when we predict the string was not in the hash table but in fact it was. We select a threshold that minimizes the total error rate, that is, the sum of the Type I and Type II errors. The threshold and total error rate is given in Table 2 for each of 1000, 2000, and 3000 strings. We see that these attacks succeed with high accuracy.

While these are the values which minimize the sum of the errors, it can be useful in situations to minimize one error at the cost of the other. A graph comparing Type I errors to Type II errors is included in Figure 1.

**Application detection through string interning.** We also analyzed eleven open-source GAE applications, to determine whether a malicious application could recognize their presence on the same JVM. Since string literals in Java are interned automatically, it is possible to determine if an application resides on the same JVM as a target by checking if the literal strings in that application have been interned. As shown in Table 3, we found that this method is likely to be very effective: each of the 11 applications interns many string literals that are not found in any of the other ten applications. For each application, we were able to identify a string literal that is not only unique among these eleven applications, but also does not appear in any other application indexed by Google Code Search.

| Application | String Count | Unique Strings |
|---|---|---|
| bdaywisher | 39 | 25 (64%) |
| birthdayplus | 553 | 438 (79%) |
| forum-botty | 152 | 97 (64%) |
| jumpnote | 555 | 249 (45%) |
| partnertracker | 1926 | 1720 (89%) |
| partychapp | 1212 | 888 (73%) |
| portexy | 56 | 33 (59%) |
| sharepie | 38 | 26 (68%) |
| thoughtsite | 927 | 820 (88%) |
| traveljournal | 124 | 99 (80%) |
| youtube-direct | 402 | 325 (81%) |

Table 3: Unique strings in 11 GAE applications.

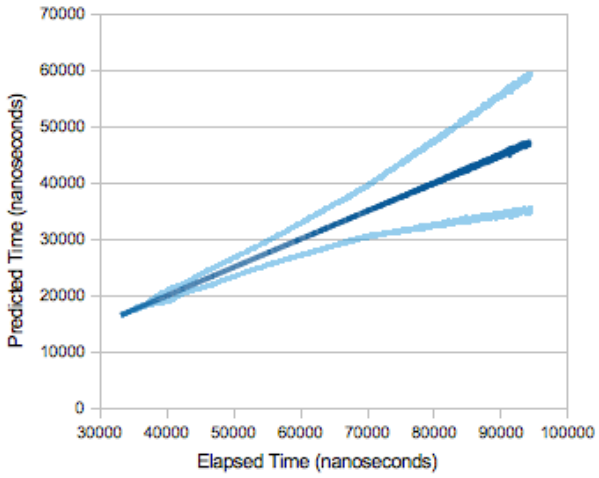


Figure 2: Predicted seeding time given elapsed time.

### 5.2 java.util.Random

We now demonstrate that an application can accurately predict the value of the static counter.

The goal of our attack is to deduce the value of the static counter. Since we can not access this variable directly, we must observe it through its impact on the seed. Our method to compute the static counter first infers the approximate time which the `Random` object's constructor received from its call to `System.nanoTime()` (hereafter called the *seeding time*). Given the seed of the `Random` object (which can be extracted using the methods in Appendix A) and the seeding time, the attacker can deduce the counter value. However, since the attacker can not accurately obtain the seeding time for any one `Random` object, he must instead create many `Random` objects and average the estimate for each to obtain a more accurate guess.

The attack attempts to form an estimate at the seeding time (which cannot be directly observed), based upon the elapsed time it takes to create the `Random` object (which can be observed). The attack consists of a training phase and an attack phase. The purpose of the training phase is to learn the relationship between the elapsed time and the seeding time. The attack phase then creates many `Random` objects, measuring the elapsed time to create each one, estimates the seeding time for each, and then forms an estimate at the static counter.

It is difficult to obtain ground truth regarding the seeding time from the `Random` object. Therefore, the training phase defines a new class `RandomAugmented`, whose source code is a copy of the Java library's source for `Random`, except that it contains an additional method to access its seeding time. The attacker then creates many ($M$) `RandomAugmented` objects, recording the elapsed creation time and the seeding time for each. For each possible value $e$ for the elapsed creation time, we filter the $M$ data points to retain only those whose elapsed creation time was $e$. For each such data point, we compute the time difference between the current time before calling `RandomAugmented` and the corresponding seeding time, averaging all of these values to obtain the mean $\mu(e)$. The function $\mu$ captures the relationship between elapsed time and seeding time. If we observe that creating a `Random` object takes elapsed time $e$, and the current time just before creating it was $t$, then $t+\mu(e)$ is our best estimate for the seeding time of this `Random` object.

During the attack phase, the attacker creates many ($N$) `Random` objects, recording for each the elapsed time $e_i$ and the time $t_i$ just before it was created. We use $t_i+\mu(e_i)$ as our estimate of the seeding time for the $i$th `Random` object, compute the corresponding estimate of its static counter, and average[5] all of these estimates. Assuming that no other application has created any `Random` objects during this time period, and correcting for the increments to the static counter due to the `Random` objects we have created ourselves, this gives us an estimate of the static counter. If $M$ and $N$ are large enough, we hope that this estimate will be accurate.

We implemented this attack on GAE servers with $M = 10^6$ and evaluated its effectiveness. Figure 2 shows $\mu(e)$ as a function of the elapsed time $e$; the lighter lines show $\pm 1$ standard deviation around the mean. We found that with $N = 10^6$ observations, the attacker is usually able to deduce the exact value of the static counter. The attack takes about 15 minutes on GAE servers, and gives a cor-

[5] Statistically, the unweighted average is not the optimal estimator in this context. In principle, it would be more efficient to compute a weighted average, with the $i$th observation receiving a weight proportional to $1/\sigma^2(e_i)$, where $\sigma^2(e)$ is the variance of the time difference for all training data points with elapsed time $e$. We found that this weighted average was less effective in practice, because our training set was too small and thus our estimate of $\sigma^2(e)$ was highly inaccurate for some values of $e$. We suspect that a more sophisticated approach might be able to eliminate this barrier and thus improve upon the unweighted average—but for our experiments, we used a simple unweighted average.

rect answer about 65% of the time. Using this attack, we observed the static counter incrementing at a rate of about 15–20 per hour (apart from our own code), which presumably indicates that other application or infrastructure code is creating about 15–20 `Random` objects per hour.

**Analysis of counter identification attack.** This attack assumes that no other code generates any other `Random` objects on the same JVM. If the attacker is on a JVM with relatively few applications, this may be possible. Otherwise, the accuracy of the attack degrades: the attack effectively infers a time-averaged value for the static counter over a 15-minute time period. The attack can be repeated multiple times for improved accuracy.

### 5.3 Math.random()

We verified that applications do request numbers from `Math.random()` by implementing our attack. We ran an application that reconstructs the seed of the `Random` object used by `Math.random()`, and thus infers the sequence of random numbers that will be generated. Then, every ten seconds we request the next random number from `Math.random()` and compare it with the expected next output in the sequence. We discovered that at random intervals, the value obtained skips values in the sequence. We infer that these gaps occur because some other application on the same JVM called `Math.random()`. This experiment shows that other GAE applications do use `Math.random()`, and that a malicious application running on the same JVM can infer or manipulate what random numbers they receive.

### 5.4 Shared Channels

To evaluate the exposure of a typical application to potential abuse by an attacker through globally accessible mutable state, we statically analyzed a set of four GAE applications. Specifically, we performed a static points-to analysis to quantify the number of application variables that may reference globally accessible objects. Since these global objects would also be available to a malicious application co-habiting the same JVM, each reference is a potential channel for information leakage, or to influence the execution of a victim application.

To perform the static analysis, we used the Chord program analysis platform [10] to produce a context-insensitive points-to relation for static, instance, and local variables for each application. We used these relations to identify each application variable that might reference a global object that can be accessed without violating the GAE class whitelist. Objects known to be immutable (e.g., `String` objects, final classes whose fields are all declared final) were excluded, since they do not leak information and cannot be influenced by an attacker. In this way, we obtained an upper bound on the number of potentially malicious shared objects.[6] The results of this analysis are shown in Table 4. We do not report shared objects referenced by static application variables, since none were discovered during the analysis.

| Application | Instance | Local | Total |
|---|---|---|---|
| bdaywisher | 397 | 0 | 397 |
| forum-botty | 265 | 0 | 265 |
| partychapp | 8414 | 29 | 8443 |
| youtube-direct | 4167 | 0 | 4167 |

Table 4: Instance and local variable references to attacker-accessible shared objects for a set of GAE applications.

Manual examination suggests that each application accesses a large number of shared, globally accessible, mutable objects. This potentially creates a large attack surface for malicious applications running in the same JVM.

## 6 Discussion and Defenses

The primary barrier to exploitation of the attacks we have found is the difficulty of arranging for the malicious application to be scheduled on the same JVM as the target application. While this might be possible, it appears to be a difficult task. Due to the sheer number of servers and the limited number of servers that any one malicious application can become scheduled on, an attacker would need to introduce thousands of malicious applications. In principle, this may be possible for a motivated attacker: applications switch servers naturally, and once an application lands on the same server as the target, switching JVMs requires little work. In practice, however, we expect such an attack would be difficult.

We present several defenses against the attacks we have found. The most important defense is to make it difficult for an attacker to control the scheduling of applications.

### 6.1 Preventing Relocation

We suggest that an application which throws an `OutOfMemoryError` not be moved to a new JVM. As observed earlier, though this error does not shut down the JVM, GAE still transfers the application to a new JVM. Changing the current behavior would make it significantly more difficult for applications to reach the same JVM as a target, even when located on the same server.

We also suggest that providers record how often applications switch JVMs and servers. The rate at which applications switch servers or JVMs could be throttled, and applications that repeatedly hit this limit could be flagged or disabled.

[6] The true set of shared channels may be smaller than that reported due to the imprecision of the static analysis and the uncertain immutability of some objects.

### 6.2 Preventing Server and JVM Detection

Our attacks rely on detecting which servers and JVMs the attacker and target are on. We suggest replacing `System.nanoTime()` with a method which, instead of returning the time in nanoseconds since the server started up, returns the number of nanoseconds since a fixed point that does not uniquely identify the server. One possible value for this fixed point could be the number of nanoseconds after the Unix epoch mod $2^{64}$, or the number of nanoseconds since the first day of the month that the server started up, also mod $2^{64}$.

Preventing JVM detection is much more difficult, since applications can mark which JVM they are on by recording the identity hashcode of objects. Since many existing applications use `System.identityHashCode`, it would not be practical for GAE to block this method. For newly designed systems, however, it would be possible to prevent calls to `System.identityHashCode`, either directly or indirectly.

### 6.3 Eliminating Shared Channels

All reachable objects shared between applications must be transitively (deeply) immutable. This prevents malicious applications from directly influencing the execution of other applications within the same JVM, although it does not eliminate the potential for more general side-channel attacks.

To prevent seed-guessing attacks, we suggest that the default constructor for `Random` should initialize its seed with a random number obtained from a static `SecureRandom` object. This would retain the speed of the current `Random` object, but prevent an attacker from determining the number of `Random` objects created in a period of time.

To eliminate the timing attack on interned strings, we propose that distinct intern pools be created for each application.

### 6.4 Application-Layer Defenses

Many applications that handle sensitive data can take steps of their own to protect this data. Applications should avoid the use of `Math.random()` entirely. Applications that create `Random` objects should do so by calling `new Random(System.nanoTime())` to avoid the static counter. Developers should also verify there is no code path through which sensitive data is interned. Finally, applications may choose to remove literal strings in order to make it more difficult for other applications to detect their presence, and instead create these strings at runtime from character arrays — since the character arrays will not be interned at runtime there would be no way to detect the application.

## 7 Related Work

Language-based isolation is not the only approach used in building cloud computing platforms. Ristenpart et al. [13] explore information leakage in Amazon EC2, a hypervisor-based platform. They built methods to map server IP addresses, information useful for then spawning a malicious VM co-resident with a target VM. They also showed how a malicious application can use a side channel to attack co-resident target apps.

Others have proposed protections for personally identifiable information (PII) in the cloud [2]. Under the threat model of a malicious provider, Gentry [4] proposes a fully homomorphic encryption scheme which allows computation on encrypted data.

The use of shared resources as side channels has also been examined on individual systems off the cloud. For example, Percival [11] uses cache misses on a multicore system to extract a RSA private key from OpenSSL.

Language-based isolation is also implemented in Microsoft's Common Language Runtime (CLR) [8], a virtual machine for the .NET Platform. Singularity, a research operating system, uses language-based isolation techniques to implement lightweight process protection without hardware [6].

Joe-E, an object-capability language based upon Java, prevents all of the attacks in this paper [7]. Our experience designing Joe-E was helpful in identifying attacks against Google App Engine.

## 8 Conclusion and Future Work

In this work, we have identified a number of potential attacks against cloud providers that use language-based techniques for isolation. We demonstrate that if an attacker can get his malicious application scheduled on the same JVM as a target application, there are a number of practical attacks that may breach the confidentiality or integrity of the target application. However, we also discovered that it seems to be difficult for an attacker to become co-resident with a target application on Google App Engine. As a result, Google App Engine seems to provide good security against targeted attacks.

One of the main lessons of this work is that the scheduling policy under which applications are assigned to servers or virtual machines plays a significant role in the security of language-based cloud computing. Our analysis suggests that, if the scheduling policy is chosen well, language-based cloud computing can provide strong protection. For instance, our work suggests that Google App Engine's scheduling policy was well-chosen. We believe scheduling policies in cloud computing are a rich area for further research. We also presented a number of other defenses to the attacks we found. We hope that our analysis will be useful to developers of future language-based cloud computing services.

**Acknowledgments**

This work was partially supported by the AFOSR under MURI award FA9550-09-1-0539, by National Science Foundation grant CNS-1018924, and by a generous gift from Google. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funders.

**References**

[1] Amazon, Inc. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`, April 2011.

[2] Angin, P., Bhargava, B., Ranchal, R., Singh, N., Linderman, M., Ben Othmane, L., and Lilien, L. An entity-centric approach for privacy and identity management in cloud computing. In *Reliable Distributed Systems, 2010 29th IEEE Symposium on* (Oct. 31 2010-Nov. 3 2010), IEEE, pp. 177 –183.

[3] Crosby, S. A., and Wallach, D. S. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium* (Aug. 2003).

[4] Gentry, C. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009. `http://crypto.stanford.edu/craig`.

[5] Google, Inc. Google App Engine. `http://code.google.com/appengine/`, April 2011.

[6] Hunt, G. C., and Larus, J. R. Singularity: rethinking the software stack. *Operating Systems Review 41*, 2 (April 2007), 37–49.

[7] Mettler, A., Wagner, D., and Close, T. Joe-E: A security-oriented subset of Java. In *Proceedings of 17th Network and Distributed System Security Symposium (NDSS 2010)* (Mar. 2010).

[8] Microsoft, Inc. Common Language Runtime. `http://msdn.microsoft.com/en-us/library/8bs2ecf4(v=VS.71).aspx`, May 2011.

[9] Microsoft, Inc. Windows Azure. `http://www.microsoft.com/windowsazure/`, April 2011.

[10] Naik, M., Aiken, A., and Whaley, J. Effective Static Race Detection for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada, 2006), ACM, pp. 308–319.

[11] Percival, C. Cache missing for fun and profit. In *Proceedings of BSDCan 2005* (May 2005), BSDCan.

[12] Rackspace, Inc. Rackspace Cloud. `http://www.rackspace.com/cloud/`, April 2011.

[13] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the ACM Conference on Computer and Communications Security (ACM CCS)* (November 2009), ACM.

## A Computing a java.util.Random Seed

Java's `Random` object is a linear congruential number generator, with constants $a = 25214903917$, $c = 11$ and $m = 2^{48}$. That is, given an initial seed $s_0$, the pseudorandom sequence is given by setting $x_0 = s_0$ and then letting $x_{n+1} \equiv ax_n + c \pmod{m}$.

Therefore, given $x_n$ we can find $x_{n-1}$ by computing $x_{n-1} \equiv a^{-1}(x_n - c) \pmod{2^{48}}$ where $a^{-1} = 246154705703781$. If we know how many iterations have been applied and know the current state, we can step the generator backwards until we find the seed.

In reality the situation is slightly more complex since the `Random` object never outputs the full 48-bit state. For instance, `nextInt()` returns the upper 32 bits of the state. To extract the full 48-bit state of the generator, we can call `nextInt()` twice. The first call reveals the upper 32 bits of $x_n$. Then, a simple brute-force search for the lower 16 bits of $x_n$ can be used to find the value which, when stepped forward once, agrees with the result from the second call to `nextInt()`.

## B Computing the Math.random() Seed

We can compute the seed of the `Math.random()` object using the attack described in Appendix A, with one modification.

`Math.random()` does not allow an attacker to obtain 32 bits of state. `Math.random()` calls `nextDouble()`, which is computed by combining two consecutive calls to the `next()` method of `Random`. The first call returns 27 bits, and the second call returns 26 bits.

Thus, the attack is carried out by using the first call to deduce the upper 27 bits of $x_n$, and then carrying out a brute-force search over the remaining $2^{21}$ possibilities to identify the one which produces the correct next 26 bits of output.

## C Distance Between Seeds

Suppose we have extracted the state of `Math.random()`'s `Random` object on one JVM, and then at some later point we want to test whether we are resident still on the same JVM. We can extract the state of `Math.random()`'s `Random` object a second time, and then test whether the second state appears in the sequence of states shortly after the first state.

In other words, we have two states $x_n$ and $x_m$, where $n, m$ are unknown, and we wish to calculate the distance $k = m - n$ between them. (If the distance $k$ is small relative to $2^{48}$, then we conclude these two states are probably from the same `Random` object, possibly with a few other calls in the middle. If the distance is large, we conclude that these might be two states from two different `Random` objects, since the only way for the `Random` object's state to change is through calling it which places a fundamental bound on the rate at which the state can change.) One approach to this problem is to try stepping $x_n$ forward, to compute the sequence $x_n, x_{n+1}, x_{n+2}, x_{n+3}, \ldots$, and check whether $x_m$ appears somewhere in this sequence. However, this is may be relatively inefficient if $k$ is large.

We show that, by taking discrete logs modulo $2^{48}$, this problem can be solved more efficiently.

In terms of $x_n$, the formula for $x_{n+k}$ is as follows:

$$x_{n+k} \equiv a^k x_n + c \cdot \frac{a^k - 1}{a - 1} \pmod{2^{48}}$$

Rewriting $x_{n+k} = x_m$ and simplifying,

$$\begin{aligned} x_m &\equiv a^k x_n + c \cdot \frac{a^k - 1}{a - 1} \\ &\equiv \frac{a^k x_n (a-1) + c(a^k - 1)}{a - 1} \\ &\equiv \frac{a^k (x_n (a-1) + c) - c}{a - 1} \pmod{2^{48}} \end{aligned}$$

Solving for $k$, we find

$$a^k \equiv \frac{x_m (a-1) + c}{x_n (a-1) + c} \pmod{2^{48}}$$

i.e.,

$$k \equiv \log_a \frac{x_m (a-1) + c}{x_n (a-1) + c} \pmod{2^{48}}.$$

Computing the discrete log is in general a difficult problem. However, computing the discrete log modulo a power of two is very easy: we can first compute the discrete logarithm modulo 8, then use that to compute the discrete log modulo 16, then modulo 32, and so on. The algorithm is as follows.

Suppose we wish to compute $L_m = \log_a y \pmod{2^m}$. We assume that a solution does exist and that $a, y$ are odd. We first compute $L_3$, then $L_4$, then $L_5$, etc., as follows. Note that $a^{2^{m-2}} \equiv 1 \pmod{2^m}$ for $m \geq 3$, so $L_m$ can be taken modulo $2^{m-2}$. Moreover, $L_m \equiv L_{m-1} \pmod{2^{m-3}}$. Consequently, either $L_m = L_{m-1}$ or $L_m = L_{m-1} + 2^{m-3}$. Given $L_{m-1}$, it is easy to try both possibilities for $L_m$ and determine which it is. So, given $L_{m-1}$, we can compute $L_m$ as

$$L_m = \begin{cases} L_{m-1} & \text{if } a^{L_{m-1}} \equiv y \pmod{2^m} \\ L_{m-1} + 2^{m-3} & \text{otherwise.} \end{cases}$$

To start, we compute $L_3$ by trying all possibilities for the exponent. Then, we apply the recurrence relation above iteratively until we have computed $L_{48}$, the distance between the two seeds.

This enables us to rapidly compute the distance between two states of a linear congruential generator, and thus to test whether two states are likely to have been produced by the same generator (possibly with several other calls to the generator in between).

## D Generating Strings with Identical Hashcodes

Java's `String` class calculates the hashcode of a string $s$ by the following formula:

$$h = \sum_{k=0}^{N-1} 31^{N-1-k} s_k \bmod 2^{32}$$

where $N$ is the length of the string and $s_m$ is the 16-bit Unicode value for the $m^{th}$ character of the string, starting with the leftmost character at index 0.

Therefore, given a string $s$ it is possible to generate a new string with the same hashcode as $s$ $k$ where $s_k$ is nonzero and creating a new string which is identical to $s$ but replaces the value at $s_k$ with $s_k - 1$ and the value at $s_{k-1}$ by $s_{k-1} + 31$. We can verify this is correct because of the identity $31 s_{k-1} + s_k = 31(s_{k-1} - 1) + (s_k + 31)$. This procedure may then be repeated to generate more strings with the same hashcode.