# Present but unreachable: reducing persistent latent secrets in HotSpot JVM

Adam Pridgen
*Rice University*

Simson L. Garfinkel
*George Mason University*

Dan S. Wallach
*Rice University*

## Abstract

The two most popular Java platforms that use the HotSpot Java Virtual Machine (JVM) are Oracle and OpenJDK. The HotSpot JVM automatically manages application memory for the developer using generational garbage collection (GC). Unfortunately, this managed memory fails to give developers the power to adequately sanitize sensitive data in objects and defend against memory disclosure attacks, i.e., attackers who can read the memory of a process containing Java objects. This problem stems from two design flaws in the GC tasks. First, the generational GC allows more than one copy of an object to exist in multiple heaps, even though they are unreachable. Second, when objects become garbage, they're not sanitized or zeroed by the garbage collector; they survive until the memory is reused for a new object. Even if a developer wants to explicitly sanitize critical data, such as cryptographic keys for closed network connections, there are no *manual* mechanisms for this activity because the data is out of the developers reach. Consequently, this sensitive data can be extracted from garbage after applying object reconstruction techniques to the heap. For this paper, we show that up to 40% of the observed TLS encryption keys can be recovered from a Java heap using Oracle's TLS 1.2 implementation. This paper also shows how modest GC changes reduce the amount of recoverable data, albeit with performance overheads as high as 50% in the worst case.

## 1 Introduction

Managed memory runtime environments like the Java platform eliminate many kinds of programming errors that can become security vulnerabilities. For example, Java programs are not vulnerable to buffer overflow attacks, making Java (and any other "safe" language) attractive for building security-critical software. Meanwhile, techniques such as just-in-time compilation, hot-spot optimization, and parallel garbage collection, have largely eliminated the performance penalty of using managed runtime environments. These features along with a rich set of standard libraries have led to a broad adoption of Java and other such languages.

However, selecting runtimes that depend on the HotSpot JVM may be a risky proposition when dealing with sensitive user or application data. Viega first identified this risk in 2001 [19]. Our research shows that the HotSpot JVM allows *latent secrets* (e.g. sensitive data found in garbage objects) to remain in the heap. Depending on the amount of memory pressure, latent secrets might stick around for a long time. Furthermore, because Java provides no direct access to the underlying memory, developers cannot *manually* sanitize their sensitive data once it expires. An attacker, on the other hand, might still be able to gain access to the raw memory, perhaps through a hypervisor bypass attack, perhaps through access to a swap or hibernation file, or perhaps through a variety of other means. Of course, we would hope that other security mechanisms can keep an attacker away from this data, but it's still highly desirable to sanitize sensitive data as soon as it's no longer necessary, minimizing the damage such an attacker might cause.

Managed runtimes typically employ two forms of automated memory management. Reference counting, which is used by runtimes like Python or Swift, maintains a *count* of handles or references to an object. When a referent acquires a reference to the object, the count is increased, and when the referent releases the reference, the count on the object is decreased. Once the count reaches zero, the object can be *deleted* or deallocated. In such systems, it would be trivial to add logic to zero out these dead objects, but Java utilizes a different form of garbage collection.

Java uses *tracing garbage collection*, which measures reachability from a set of *root objects* to every object in memory. When an object can no longer be reached from the root set, the object is considered *garbage*. Modern

GC performance is heavily dependent on *laziness*. There will always be a gap between the time an object becomes unreachable and the time the garbage collector notices and ultimately reuses that memory. The GC can also *rearrange* the memory heap to help improve collector performance and reduce pause times. This rearrangement process inherently involves *copying* objects, which can leave behind multiple "old" copies that will not be immediately zeroed. Our findings are similar to other work (e.g. [3, 7]) that demonstrate potential confidentiality failures because of a semantic gap between the language that programmers use, the implementation of that language, and its underlying execution environment.

In this paper, we establish the volume of secrets that an unsanitized heap can expose, using a TLS web client atop Oracle's HotSpot JVM, driven by a synthetic load under different levels of memory contention. We capture whole system memory images and then use binary string searches to find TLS keys and other sensitive data, no matter where in memory they may be, whether they're reachable or not. We then made changes to the TLS code and the garbage collector in attempt to eliminate these secrets. Our benchmarks show a worst-case hit of 50%, with common cases being more reasonable.

## 2 HotSpot Memory Background

Java, and systems like it, use garbage collection (see, e.g., [5, 12]), to manage their memory. A common strategy in most GC implementations is to support *generational copying*, where new objects are allocated in a *young generation* and are later promoted to a *tenured generation* if they're still reachable. The HotSpot JVM implements several different garbage collectors, all of which follow some variation of this strategy. Figure 1 shows a typical Java heap memory layout.

The young generation is partitioned into an *eden space* where objects are created (and where most die [13]), and two *survivor spaces* that hold objects that are copied out of the eden space. The eden space is further partitioned into *thread local allocation buffers* ("TLAB"), where allocation is performed using a "bump-the-pointer" technique which minimizes the number of locks required for multi-threaded applications. As objects age and survive GC, they are generally migrated from the eden space to the survivor spaces and are then tenured if the objects either surpass an age threshold or if the young generation runs out of memory. Because of its focus on performance, the JVM does not clear the contents of memory when an object is moved from one generational area to another [18]. Stale data will eventually be overwritten when one of these memory spaces is reused, but this won't happen immediately.

Most garbage collectors can take advantage of additional RAM, gaining additional performance when faced with less pressure to compact live objects and reuse memory. Consequently, as the heap size grows, there is a higher probability that latent secrets stick around longer and can be recovered from RAM, even though they're no longer "reachable" by the original program. While it might seem tempting to solve the problem by artificially capping the size of the heap, and thus forcing more memory reuse, such a drastic tactic would lead to serious performance problems.

Consequently, this research digs deeply into how the HotSpot JVM's memory system operates. To a reader with a general familiarity with garbage collection but no specific familiarity with the HotSpot JVM, the most surprising aspect of the JVM is its use of a region-based allocator. These regions can be used for a variety of things outside of the standard heap, like Java class metadata. They are also allocated by the garbage collector and used for its various spaces. This means that we must look beyond the garbage collected heap to understand how latent secrets might land in other memory regions.

## 3 Prior Work

In 2001 Viega identified that memory is not cleared when it is deallocated in C, C++, Java, and Python runtimes [19]. Chow et al. showed that Unix operating systems and standard libraries failed to sanitize memory when it was deallocated. Attackers could exploit this issue to recover latent secrets from common applications like Apache and OpenSSH. The authors implemented proper sanitation in the Unix operating systems with roughly a 1% impact on performance [3, 4]. However, Chow et al.'s techniques cannot address the latent secrets found in the JVM. Remember, the JVM uses it's own memory management primitives, and GC only exacerbates this problem in the Java heap.

The process of extracting latent secrets from dump files or system memory seems challenging, but many researchers have found the task to be quite surmountable. For example, Harrison and Xu identified RSA cryptosystem parameters in unallocated memory that had been inadvertently written to untrusted external storage as the result of a Linux kernel bug [9]. Halderman et al. showed that AES encryption keys can be trivially detected in RAM from their key schedule [8]. Case presented an approach for analyzing the contents of the Dalvik virtual machine [2]. Similar attacks are possible against Android smartphones, allowing for the recovery of disk encryption keys [14] and Dalvik VM memory structures [10]. Jin et al. used symbolic execution and intra-procedural analysis to accurately extract the composition of type data generated by C++ programs [11].

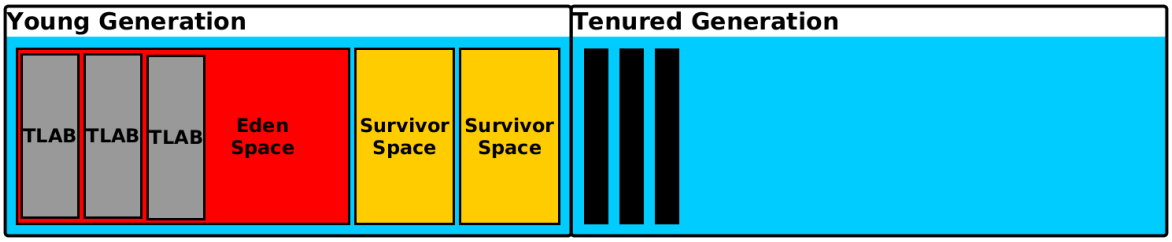



Figure 1: A typical generational heap layout used by the JVM, with multiple "thread local allocation buffers" (TLABs) in the eden space. Surviving objects are eventually relocated to one or more "survivor" spaces before finally landing in a "tenured" space.

Finally, there are a variety of memory disclosure attacks and techniques. The most straightforward technique uses one process to read the memory of another process utilizing a suitable device driver or kernel module (e.g. */dev/mem* or the */proc/nnn/mem* devices). Because such devices are commonly exploited by malware, many operating systems no longer include devices for reading the memory of other processes. Until recently, employing memory-acquisition kernel modules on a running system was complicated. Compilation of kernel modules required the specific source code used to create the target system's kernel. Stüttgen and Cohen overcame this obstacle, developing an approach for safely loading a pre-compiled kernel modules into memory on running Linux systems [17]. This approach is now used by the Rekall Memory Forensics Framework [6].

Halderman et al. developed the "cold-boot attack" in which the DRAM memory from the target computer is physically chilled and then transferred to a computer that is known not to wipe memory on boot [8]. It is possible to physically read the contents of a computer's memory using hardware that provides direct memory access (DMA). Consumer firewire interfaces, JTAG interfaces, or even specially constructed interface cards use DMA, making this vector quite pervasive. VöMel and Freiling survey these and other techniques for acquiring main memory in computers running the Windows operating system [20]. Consequently, the threat of an attacker conducting a memory disclosure attack is significant, justifying efforts to mitigate these attacks.

We note that this class of attack may apply in a variety of different devices. Smartphones and laptops may be physically stolen or otherwise captured, giving a motivated attacker physical access to the device. Cloud servers (e.g., Amazon's EC2) may run in virtual machines that can migrate from system to system, allowing for a variety of attacks such as capturing a system image while it's migrating, or accessing the system's memory from a potentially compromised hypervisor. This research is predicated on the assumption that an attacker has somehow found a way to capture a system memory image. We will now explore how much damage this class of attacks can cause.

## 4 Measuring Latent Secrets

This section describes the experiments, infrastructure, and software we used to measure latent secrets in the HotSpot JVM. For simplicity, we chose *black-box* analysis techniques. We measure latent secrets by externally capturing important inputs (e.g., TLS key data) and then scanning system memory dumps of a TLS web client for those data.

Our approach is simple and reliable. Consider the alternatives: we might instead try to perform dictionary attacks on TLS sessions, or we might try to use one of the "integrity-only" TLS ciphersuites. But why do that when we can just build a custom TLS server that logs all of the relevant key material for our search? Similarly, we could try to engineer a search through the reachable live memory of the JVM, but this would miss the latent secrets we're interested in, and would also miss any buffers allocated through the "region" system that's used below the garbage collector (see Section 2).

In our experiments, we'll be looking at memory dumps of a Java TLS client. If we were to instrument it in order to capture TLS keys as they're used, our instrumentation might perturb the system that we're trying to measure. Consequently, we instead instrumented a TLS web server, built from the OpenSSL library. Our client-side Java application makes TLS connections, over and over, to our instrumented web server, creating an abundance of latent secrets in the heap. On the web server, the modified OpenSSL library records each session's key material (i.e., the pre-master secret (PMS) and the master secret (MKB)). Given a memory dump of our Java client's Linux virtual machine, we simply search the entire memory dump for all of the observed key material from every TLS session.

We run these experiments on a small cluster of PCs

running Linux KVM. We limit the number of simultaneous virtual machines on a given machine to avoid resource contention, which might otherwise impact our performance measurements. Ultimately, we run only one TLS client per physical machine at a time. Furthermore, we reboot each Linux virtual machine after each experiment completes, allowing us to restart each run from an identical starting point, including all user and kernel-level state.

Our experiment infrastructure is composed of several virtualization servers consisting of over 8 logical cores and at least 32 GiB of system RAM. The servers all run Ubuntu 14.10. Two of the servers are designated to run the virtual machines executing the Java client and two others are used to run the web servers. Additionally, two machines are used solely for performing the memory analysis, which consists of identifying and pin-pointing TLS keys, string extraction, and aggregating the analysis of each experiment. These machines have 24 logical cores with either 96 GiB or 384 GiB of RAM. The larger machines were necessary to efficiently post-process the large memory dumps.

Our TLS web server and synthetic client both run on an x64 Ubuntu 14.04 LTS virtual machine. The web server uses NGINX and TLS 1.2 to serve several static web pages. The web server VMs utilize four logical cores and 2 GiB of RAM—enough to ensure that server performance wasn't the bottleneck for our client measurements. The virtual machines that run the Java clients have several different configurations, such as varying the maximum heap size, to induce different amounts of memory pressure.

## 4.1 Synthetic Client Functionality

Our synthetic Java client is best described as a *multi-threaded*, *configurable* TLS web client. The client uses several parameters that allow us to manipulate the memory pressure exerted on the heap, the number of concurrent threads, the maximum number of HTTPS requests, and the lifetime of a thread sending the web requests. These parameters give us the ability to model basic transactions for applications like a thick- or web-service client. However, for the purposes of this paper's experiments, we choose two specific memory pressure configurations and measure the latent secrets produced in the Java process.

In particular, we use a configuration that creates as much heap memory contention as possible in the JVM and then another that attempts to keep contention low. Both configurations allow up to 192 concurrent TLS connections that are active for at least 96 seconds. The two configurations differ in amount of heap memory allowed that can be allocated from the JVM, and they also differ

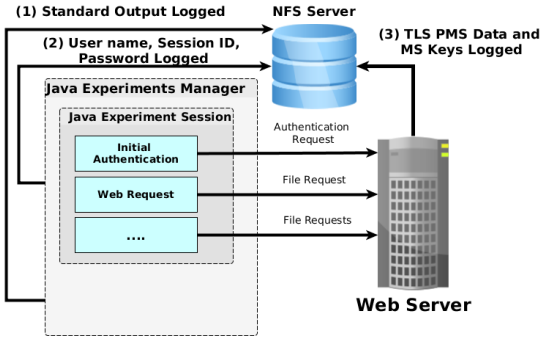



Figure 2: Functional overview of our synthetic web client.

by the number of requests allowed per thread. The high memory pressure (HMP) experiment allows for the allocation of at most 80% of the JVM's heap memory and allows up to 192 requests per thread, The the low memory pressure (LMP) experiments only allow a maximum of 20% of the JVM's heap memory and up to 48 requests per thread.

Figure 2 shows how the two main components of the synthetic client work. The first component, Java Experiment Manager (JEM), manages all the experimental sessions (threads). The second component, Java Experimental Sessions (JES), implements the web client functionality in a Java thread. The JEM is responsible for managing the number of JESs and enforcing the experimental behavior and garbage collection parameters. A Python script starts the experiment with parameters that define the experimental behavior of the synthetic client, the IP address of the server, and the place to store log files containing events and other data.

Each JES is started by the managing JEM, using the behavior and garbage collection parameters given to the JEM. The JEM controls the number of concurrent JESs, JES allocation behavior, JES HTTPS requests, and the overall lifetime of the JES thread. Parameters controlling the garbage collection define the frequency of collection, when to start collecting, and whether or not to pause JESs after the first GC. Our experiments also allow us to vary the TLS library in use (Oracle vs. BouncyCastle) and to use the Apache `HttpClient` vs. a basic TLS socket connections.

We implement and use three different TLS web clients in the JES. The most basic TLS client is the Socket TLS client. This type of client opens a TLS socket to the remote server, sends the raw HTTP request in a formatted string, receives the data, and then closes the socket. The second client uses the Apache "HTTPComponents" library, to create an `HttpClient`, which then connects to the remote host in the *url* and retrieves the *uri*. Most

of the internal HTTP mechanics are abstracted away, simplifying the entire retrieval task. However, this abstraction removes sensitive data like usernames and passwords from our control, which we hypothesize might lead to a larger volume of latent secrets. The final client type is a variant of the Apache `HttpClient` that uses the BouncyCastle cryptography library instead of Oracle's cryptography library. This option allows us to measure whether the TLS implementation, itself, can contribute to the volume of latent secrets.

Each implementation makes every effort to remove excess references and prepare the connecting object for a future collection. In the Socket TLS Client, we close the `Socket` and set our references to it to `null` as soon as possible. The Apache `HttpClient` does not have an explicit close or shutdown API, so we can only set our reference to it to `null` and hope the internals aren't hanging onto anything. We also note that the Apache `HttpClient` uses an `HttpClientConnectionManager` to manage client connections. This manager may choose to maintain open connections to the remote hosts. This socket reuse makes reconnecting to an old peer much faster, avoiding the overhead of rebuilding a TLS connection, but also keeps key material live in memory longer.

## 4.2 Memory and data analysis

Data analysis and extraction happens in three distinct phases. After an experiment, the resulting memory, TLS session data, and web client logs containing sensitive HTTP parameters like the username and password pair, etc. are queued for analysis. First, the analysis process scans the memory dump for latent secrets (e.g. PMS and MKBs) using `jbgrep`. This scan is conducted using two perspectives of the memory dump. The first perspective is the *raw* memory dump, which reveals all the latent secrets along with a count for each one found. The second perspective reconstructs the process memory using a virtual memory mapping, which informs us where the latent secret exists in the Java process (e.g., which generational heap space or the address in the Java process).

After all the latent secrets are identified and counted, a post-processing step enumerates every HTTP request for each JES and pairs these requests using the TLS session data. These pairings are made based on time. Specifically, we use monotonically increasing timestamps to pair the sessions. We are unable to pair the exact TLS session to the corresponding web request, but this granular knowledge is not necessary to create an approximate timeline showing live objects versus latent garbage in the heap.

## 4.3 Experiments

Here we provide an overview of experiments and the overall goals, but we describe the exact configuration of the experimental parameters alongside the description of each experiment in Section 5. Each experiment explicitly configures the JVM to use a serial garbage collector, to divide the young and tenured generation equally, and to preallocate all memory from the operating system. We configure the JVM in this manner to reduce the amount of variability from one experiment to the next. For example, preallocating memory and specifying the size of the young and tenured generation allows us to efficiently map latent secrets into specific heap regions. Otherwise, we would need to extract and parse GC logs or JVM memory structures to learn the actual heap layout.

We performed four different experiments. The first experiment measures the persistence of latent secrets in a web client using an unmodified Oracle HotSpot JVM with the goal of saturating the heap with latent secrets. In this experiment, a HMP configuration using either the Socket TLS Client and the Apache TLS Client without BouncyCastle are used. We use the measurements from the first experiment to develop mitigations that zero out latent secrets.

The next three experiments assess whether or not our mitigations work. The fixes include patches to the Java Cryptography Extensions (JCE) and Java Secure Sockets Extensions (JSSE) libraries and modifications to the default generational serial garbage collector. We leverage the OpenJDK source code to help us with our analysis and make the changes as necessary. In both cases, we aim to reduce the number of latent secrets through explicit sanitation of primitive data arrays.

Experiment Two compares the modified OpenJDK HotSpot JVM and the patched runtimes to an unmodified Oracle HotSpot JVM. We conduct the experiment with three different runtime configurations in conjunction with our three different TLS clients. Using information from the first experiment, we wait until at least 10K TLS sessions have been observed to ensure that the 4 GiB heap is saturated.

Experiment Three compares the modified OpenJDK HotSpot JVM to the Oracle HotSpot JVM. This experiment duplicates most of the memory parameters from the first experiment; we use a reduced number of iterations for the modified HotSpot JVM because of the low variance in the number of latent secrets.

Experiment Four focuses on eliminating all the latent secrets from the heap. In the second and third experiment, we observe latent secrets in the tenured space, and normally, these items are only collected if there is an allocation failure in the tenured space. We conduct this experiment by stopping the on-going experiment after a

| Heap Size (MiB) | Socket TLS Sessions | Socket TLS Keys Recovered | Apache TLS Sessions | Apache TLS Keys Recovered |
|---|---|---|---|---|
| 512 | 5186 | 489 | 7005 | 286 |
| 1024 | 4943 | 1059 | 6762 | 499 |
| 2048 | 9949 | 1845 | 13997 | 929 |
| 4096 | 9948 | 3177 | 13364 | 1608 |
| 8192 | 14942 | 4786 | 19497 | 3008 |
| 16384 | 23491 | 9058 | 34088 | 5354 |

Table 1: The average number of TLS sessions and recoverable unique keys for two different TLS clients.

specified number of TLS sessions have been observed. GC is then performed at regular time intervals for a specified number of iterations.

# 5 Removing Latent Secrets

Figure 3 shows the results from the initial assessment. This experiment uses the Oracle HotSpot JVM to execute two different web clients employing a high memory pressure (HMP) configuration. We see that the JVM retains a large number of latent secrets as the heap size becomes sufficiently large, making it a viable target for memory disclosure attacks. For each TLS key recovered, there are roughly 1-2 copies of the pre-master secret (PMS) data and 3-4 copies of fully intact master key blocks (MKB), i.e., TLS session keys. Multiple copies of key data are the result of extraneous copies and excessive references to these copies. When our results refer to "unique keys," we note that an MKB can be derived from a PMS, so if we find both, we'll only count them as one "unique key."

Where are these key copies coming from? A cursory inspection of the OpenJDK Java JDK source code reveals that local variable references are not set to `null` and *cloned* `byte[]` values are not zeroed when they are no longer needed, so the latent data will certainly stay in memory until the garbage collector reuses it. And, because of the generational structure of the GC, there may be additional copies of older keys.

Table 1 shows the average number of recoverable unique keys in each heap size. The table also shows the number of TLS session opened and closed by the Socket and Apache TLS Clients. Unsurprisingly, the number of latent secrets nearly doubles as the heap size doubles. The Apache TLS Client has notably fewer recoverable keys than Socket TLS Client, which we believe is attributable to the larger memory footprint of each `HttpClient`. The Socket TLS Client requires less heap memory per connection because it's a single socket that closes after each session.

Based on these experiments, it's quite clear that latent data is a serious concern. When key material from thousands of closed connections sticks around in memory, that radically increases the exposure of that key material to compromise.

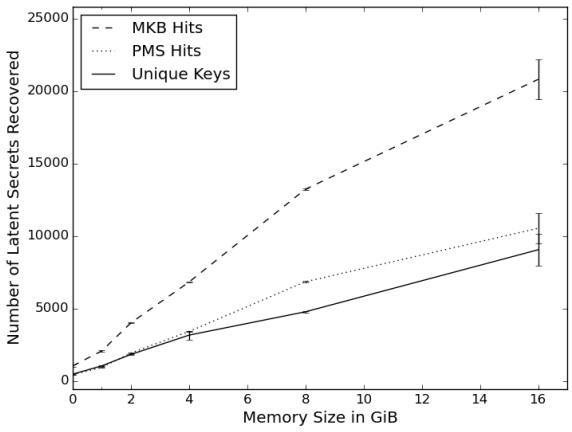
Consequently, we devise two approaches to address this issue, both requiring changes to the OpenJDK source code. First, we patch the JCE and JSSE to zero-out sensitive data when it's no longer necessary, based on manual code audits. Second, we modify the JVM internals and explicitly zero out data pages on all deallocations and unused heap after GC. Obviously, the second tactic will have a significant performance cost, but it's useful to measure how much.

## 5.1 Patching the JCE and JSSE

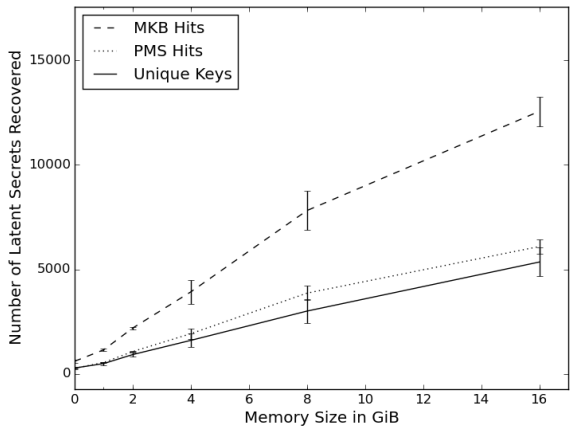
Our code audit revealed several faults that we try to address. First, the JCE and JSSE neglect to implement the `javax.security.auth.Destroyable` interface and call the `destroy` API in the `SecureKeySpec` implementation class. The whole purpose of this API is to assist developers with zeroing unneeded key data. Second, the JSSE cannot *signal* the JCE implementations when a socket closes. Closing a socket should force the destruction of TLS sensitive key material. OpenSSL, as a brief comparison, sanitizes all *dead* key data and derivative material.

To mitigate these issues, we first added code to the `SecretKeySpec` in the JCE that sanitizes key data by *default*. Next, we modify the `TlsKeyMaterialGenerator`, `TlsMasterSecretGenerator`, and `TlsPrfGenerator` classes. These changes focused on sanitizing local variables used for PMS and MKB data. Our modifications prove to be challenging, because there are no explicit contracts between callers and callees defining how to handle sensitive key material. For example, methods may clone `byte[]` objects and then pass the reference onto other methods. These references are used later by implementation classes, with no way to track when they become unnecessary to keep around. Consequently, we modified JSSE by creating a method in `SSLSessionImpl`. When the socket closes, the `SSLSocketImpl` uses a callback to destroy the MKB data in the `SSLSessionImpl`[1]. Unfortunately, after testing our changes, we found that the modifications were ineffective. We only made these measurements in conjunction with JVM-level changes (described in the next section), but our results clearly show the impact of

[1]We note that Java normally uses digital signatures to protect the integrity of its crypto libraries. We defeated this by breaking the signed code verifier. A production system, obviously, would need to be properly signed along with the JDK distribution.




(a) Latent secrets from Socket TLS Client




(b) Latent secrets from Apache TLS Client

Figure 3: These plots shows the average amount of sensitive data in the form of pre-master secrets (PMS), master key blocks (MKB), and recoverable unique TLS keys from a set of heap sizes.

these changes is quite small. The number of recoverable latent keys slightly *increases* when we try to zero things out from the JCE source code. These results are shown in Table 2.

## 5.2 Adding Sanitation to the JVM

We modify the OpenJDK HotSpot JVM source code to implement a *global* sanitation solution in the heap. For simplicity, we chose to modify only the *serial* garbage collector implementation—the default garbage collector that's used with the HotSpot JVM. First, we focus our efforts on cleansing the young generation, and then we tackle the problem in the tenured generation.

The default HotSpot GC is generational, but does a variant of mark-and-sweep in the tenured generation searching of each memory block We tagged along with the sweep phase, zeroing out regions of the memory corresponding to dead objects. When compaction is happening, we can similarly zero out the original objects, one by one, after they're relocated. (Unfortunately, the Hotspot VM's GC doesn't ever do a giant copy-compaction during collection in the tenured space with a from-space and a to-space, so there's never a huge block of memory we can blindly zero out.)

Zeroing individual objects, or arrays, as the sweep phase or copy phase figures out that they're garbage, seems like a relatively efficient change to make to the garage collection, since the memory in question was just recently touched, so should already be in the CPU's cache. Unfortunately, this strategy requires us to understand all of the specific tricks that the garbage collector uses, so we know when it's truly safe to write zeros into memory.

Notably, we encountered cases where *invalid* dummy objects were placed in the heap. Without knowing this fact, we would check pointers and class types using internal APIs, and these checks caused segmentation faults in the JVM. After some investigation, we discovered that this issue was the result of a hack to make the heap appear to be contiguous during collection, which is a precondition to make GC work correctly. Recall, each thread-local allocation buffer (TLAB) is a small partition of the eden space, so the JVM fills the empty spaces with dummy objects during GC or when a TLAB is invalidated. We work around this issue by ensuring `Klass` pointers (i.e., pointers to the C++ representation of a Java class) fall inside the Java metaspace, where all Java meta-objects (e.g. classes, methods, etc.) reside.

We also had a variety of other minor issues. For example, dealing with primitive Java types (like `byte` arrays) versus class types (like `Byte` arrays) led to confusion in our analysis. We had to add specific logic to deal with many such cases.

Modifying the garbage collector, as described above, was necessary, but we still found more latent key material that survived, and it was outside of the garbage-collected heap. As we described earlier, the JVM maintains memory blocks that can be explicitly allocated and freed. Latent secrets were getting copied there as well. We addressed the problem by sanitizing all internal memory deallocations (similar to [3]). We leverage the JVM's native memory tracking (NMT) for this task. Typically, NMT is used to track internal memory allocations to help with profiling, diagnostics, and debugging. For our pur-

| JVM Version | Keys recovered after GC: Bouncy Castle | Apache | Pure TLS Sockets |
|---|---|---|---|
| Low Memory Pressure (LMP) Results | | | |
| Oracle JVM | 1542 ± 92 | 2972 ± 81 | 1084 ± 84 |
| Modified JVM | 341 ± 55 | 827 ± 30 | 304 ± 117 |
| Modified JVM/JCE | 364 ± 102 | 848 ± 44 | 371 ± 89 |
| High Memory Pressure (HMP) Results | | | |
| Oracle JVM | 1671 ± 86 | 3052 ± 60 | 1202 ± 86 |
| Modified JVM | 406 ± 87 | 944 ± 78 | 371 ± 94 |
| Modified JVM/JCE | 375 ± 103 | 1010 ± 55 | 387 ± 56 |

Table 2: The number of unique TLS keys, exploiting all latent secrets in the heap, that are recoverable after garbage collection in the LMP and HMP experiment configurations. Three different TLS client implementations are compared using three different JVMs with low memory pressure.

poses, we use NMT to identify the size of each allocation, and then we zero the buffer before the memory is returned to allocation pool. The NMT documentation states that it produces a 5-10% performance penalty [15]. We'll discuss our performance measurement findings in Section 5.4.

## 5.3 Sanitation Effectiveness

As described in Section 4.3, we conducted four different experiments. Our first experiment represents a control group, with an unmodified JVM.

Our second experiment uses three different Java runtime configurations, namely the unmodified Oracle HotSpot JVM, the modified OpenJDK HotSpot JVM, and the modified OpenJDK HotSpot JVM with the modified JCE, JSSE, and `rt.jar`, and we run our three different TLS client applications on top of these configurations. The experiment is set-up to perform a single GC using `System.gc()`, and then all the JESs are paused until the system memory gets dumped. This explicit GC event allows us to evaluate whether or not the latent secrets are wiped from the heap. This second experiment uses a heap size of 4 GiB with four logical cores available to the Linux VM. The experiments all ran for about 20 minutes, and we collected 10 memory dumps per client and configuration. In this time period, we observe 11.9K TLS sessions using the Socket TLS Client, and the experiments using Apache TLS Client and Apache TLS Client with BouncyCastle average 16.6K and 16.8K respectively.

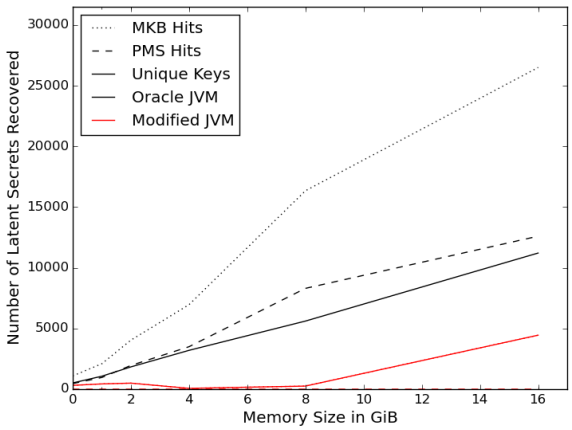
Table 2 shows an overview of the results for the Oracle HotSpot along with the two configurations using the modified OpenJDK HotSpot JVMs. For both the LMP and HMP configurations, there is a significant drop in the number of latent secrets present. However, this massive reduction is mostly attributable to the sanitation we added to the young generation's memory sweeping. Since sanitation of the heap generations depends largely on allocation failures, the tenured generation needs more collection activity to trigger the removal of latent secrets.

We also see that the JCE and JSSE modifications modestly increased the number of latent secrets in the heap. We're not entirely sure why this occurred. It's possible that our code modifications had other impacts on the code generation and GC behavior that we couldn't predict. These negative findings do reinforce the importance of support from the garbage collector. Purely application-level zeroing of data isn't going to adequately address the problem.

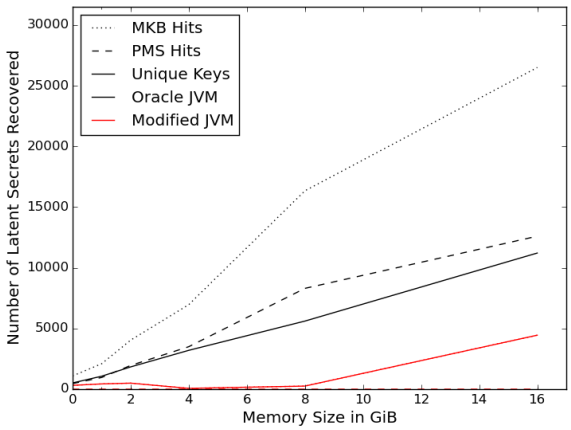
We're intrigued by the differences between the "BouncyCastle" configuration and the "Apache" configuration. Both are using the same Apache HTTP client library, so the only significant difference is that the "Apache" configuration is using the Oracle TLS library. Why is the BouncyCastle version doing so much better? A manual inspection of the BouncyCastle code shows that they're being very disciplined about their key management; they make fewer copies. That said, the "Pure TLS sockets" experiment drops the Apache HTTP client library and directly drives the Oracle TLS libraries. This gives up the performance and concurrency features of the Apache library, but it has the fewest latent secrets. These results suggest that complex interactions between libraries and networking layers can have unforseen increases in the volume of latent secrets.

In our third set of experiments, we recreate the conditions from our initial evaluation to compare the overall reduction of latent secrets. This time, we vary the JVM heap size from 512 MiB to 16 GiB on a Linux VM utilizing four logical CPUs, and we only compare the Apache and Socket TLS Client with LMP and HMP configurations. Figure 4 shows the results of these observations. The number of recoverable unique keys is dramatically smaller, as before, but in some circumstances the volume of latent secrets stays small regardless of heap size, while in other circumstances the volume of latent secrets starts small, but with very large heaps it grows significantly. We believe this is a consequence of the tenuring process. Once an important secret is tenured, it's unlikely to be noticed again by the garbage collector due to inadequate memory pressure that would otherwise force the garbage collector to process the tenured blocks of the heap.

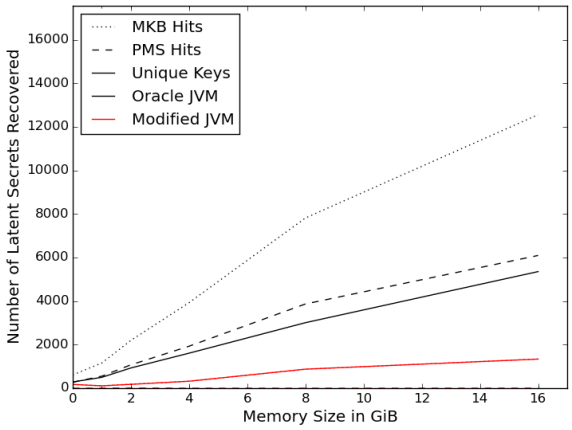
We performed a follow-up experiment to determine if it is possible to forcibly cleanse the heap. Figure 5 shows the number of explicit GC calls required to sanitize the heap. Here we use our modified OpenJDK HotSpot JVM with a 4 GiB heap size with the LMP Socket TLS client.



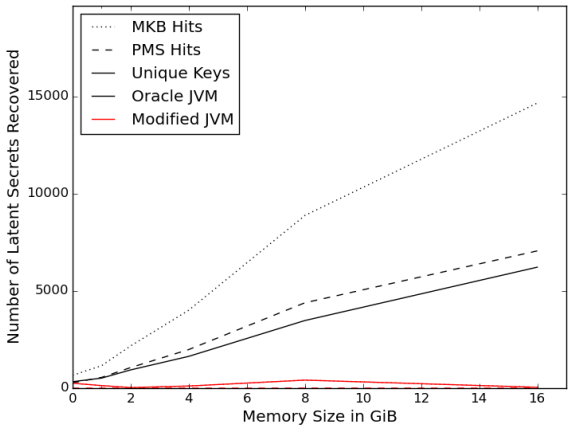

(a) Comparing Recoverable TLS Keys in Latent Secrets from the Socket TLS Client with a HMP web client




(b) Comparing Recoverable TLS Keys in Latent Secrets from the Socket Client with a LMP web client




(c) Comparing Recoverable TLS Keys in Latent Secrets from the Apache TLS Client with a HMP web client




(d) Comparing Recoverable TLS Keys in Latent Secrets from the Apache TLS Client with a LMP web client

Figure 4: These plots compare the results for the Socket TLS Client the Apache TLS Client. The lines show how many latent secrets can be removed from memory by sanitizing the heap space after garbage collection. High- and Low- pressure applications are also shown.

We control for the number of GC iterations and then timing between each collection. Our experiment waits to observe 12K requests before pausing all the JES threads and starting the GC with a call to `System.gc()`. With this modification, zero to four more iterations of GC happen at regular intervals of 30 milliseconds, 10, 20, and 30 seconds. Four GC iterations appear to do the job. We also performed this experiment with the Apache TLS Client with four iterations, and the outcome is the same.

Overall, the findings from each of the experiments demonstrate the difficulty of managing and eliminating sensitive data in a managed runtime like Java and the JVM. When the JVM does not perform heap sanitation, many latent secrets will persist for an indefinite time period. This problem only gets worse if third party libraries are used, because control over the sensitive data is relinquished to these libraries. Furthermore, there are never any guarantees that libraries will be good stewards in managing the data. Hence, even if the JVM sanitizes all data, there will always be the risk that third party code will run afoul of the developer's efforts.

## 5.4 Benchmarking

To measure the impacts of our modifications, we use two benchmarks. First, we use the DaCapo benchmark suite,

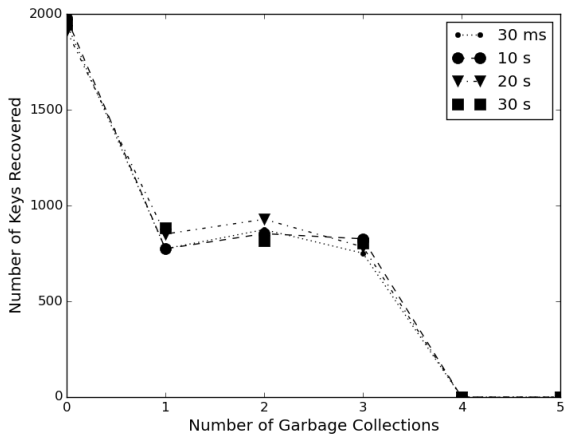



Figure 5: After the number of garbage collections required to clear the heap (4) is reached, no latent secrets can be recovered from memory. The time intervals used are 30 ms., 10, 20 and 30 seconds. Changing the interval between GC resulted in no significant change.

| Application | Allocation Failure Events | `System.gc` Events | Total Events |
|---|---|---|---|
| | 1GiB RAM | | |
| h2 | 1 | 1 | 205 |
| jython | 3 | 1 | 402 |
| lusearch | 12 | 1 | 1300 |
| tradebeans | 5 | 1 | 605 |
| tradesoap | 10 | 1 | 100 |
| | 4GiB RAM | | |
| h2 | 0 | 1 | 101 |
| jython | 0 | 1 | 101 |
| lusearch | 3 | 1 | 400 |
| tradebeans | 1 | 1 | 204 |
| tradesoap | 2 | 1 | 305 |

Table 3: Garbage collection information for each benchmark application: number of Java object allocation failures (triggering a GC), number of GC events triggered using `System.gc`, and the benchmark's overall number of GC events.

version 9.12 [1] to provide an overall assessment of our modifications. DaCapo provides a number of different applications, but we choose to use only five. We also use our synthetic client to compare the request throughput of an unmodified OpenJDK HotSpot JVM and our modified OpenJDK HotSpot JVM. We decide to include this comparison, because the DaCapo benchmark reveals some potential deficiencies in our modifications. We want to see if these impacts are observable in our TLS client using some of our past experimental parameters.

We use only five applications from the DaCapo framework, because these are the only applications that experience at least one GC event when the heap size is 1 GiB. Before we continue, we provide an overview of each benchmark program. `lusearch` is a text searching service. The `tradebeans` and `tradesoap` are day trader applications that perform workloads on a server and client, respectively. `h2` is a database implemented in Java, and `jython` is an off-shoot of Python that is interpretted and executed on a JVM. Separately, each application provides a reasonable evaluation of the performance impacts.

The DaCapo benchmarks run inside of a virtual machine, so we perform 100 benchmarks to prevent jitter and improve accuracy. We also use four different virtual machine configurations (one and four CPUs, with 1 GiB or 4 GiB of heap) with three different HotSpot JVM implementations: unmodified OpenJDK HotSpot, Oracle HotSpot, and the modified OpenJDK HotSpot JVMs. The unmodified OpenJDK HotSpot is built on the same machine using the same build settings our modified OpenJDK HotSpot JVM, which help eliminate any build or code optimization variables that might influence the benchmark results.

Table 3 shows the number of GC events per application for the two different heap sizes, and Table 4 shows the average execution time for each benchmark. These two tables show that adding sanitation to the GC workload is going to decrease performance. Aside from this point, we are unable to discern any other notable issues. `lusearch` does experience some significant performance impacts, but these impacts are due to how much memory contention the program creates. Specifically, in a time window of 1.3–4.1 seconds, each application benchmark handles 12 or 3 allocation failures on average, depending on memory of 1 GiB or 4 GiB. Looking specifically at the benchmarks with four CPUs, each individual benchmark writes roughly 6 GiB of zeros to RAM in the span of 1.3 seconds, while executing the programs actual function.

Realistically, our synthetic client behavior matches most closely to `tradesoap`. This program is a web client that sends SOAP requests to a web server and waits for a response, which is what our client does. This program experiences roughly 2 or 10 GC events on average over the course of roughly 10 seconds for the four CPU benchmark with a heap size of 4 GiB and 1 GiB. The modified OpenJDK HotSpot takes about 10 and 11 seconds for the two different cases. When there is a real threat of attack and the value of the information very high, the 1-2 sec-

| Application | Java VM (A) | (B) | (C) | (D) |
|---|---|---|---|---|
| One vCPU with 1GiB RAM | | | | |
| h2 | 3353 ms | 15.4% ↓ | 2.5% ↓ | 22.6% ↓ |
| jython | 2650 ms | 42.4% ↓ | 11.2% ↑ | 1.2% ↑ |
| lusearch | 4132 ms | 6.4% ↑ | 1.7% ↓ | 32.0% ↓ |
| tradebeans | 10403 ms | 14.6% ↑ | 17.2% ↑ | 3.1% ↑ |
| tradesoap | 17822 ms | 20.2% ↑ | 9.3% ↑ | 0.5% ↑ |
| One vCPU with 4GiB RAM | | | | |
| h2 | 3326 ms | 9.6% ↓ | 0.2% ↑ | 10.4% ↓ |
| jython | 2436 ms | 14.5% ↑ | 1.3% ↑ | 29.5% ↓ |
| lusearch | 4115 ms | 5.1% ↓ | 9.7% ↓ | 32.9% ↓ |
| tradebeans | 10242 ms | 18.7% ↑ | 24.5% ↑ | 3.5% ↑ |
| tradesoap | 17527 ms | 5.6% ↑ | 3.7% ↓ | 8.9% ↓ |
| Four vCPUs with 1GiB RAM | | | | |
| h2 | 7115 ms | 12.3% ↑ | 10.8% ↓ | 3.3% ↓ |
| jython | 2290 ms | 48.1% ↓ | 4.5% ↓ | 21.0% ↓ |
| lusearch | 1278 ms | 4.2% ↓ | 9.7% ↓ | 53.2% ↓ |
| tradebeans | 9836 ms | 27.5% ↑ | 2.3% ↓ | 8.0% ↓ |
| tradesoap | 9677 ms | 7.8% ↑ | 5.7% ↓ | 14.7% ↓ |
| Four vCPUs with 4GiB RAM | | | | |
| h2 | 7688 ms | 23.9% ↑ | 4.1% ↓ | 2.7% ↓ |
| jython | 2458 ms | 9.8% ↑ | 8.3% ↑ | 6.1% ↓ |
| lusearch | 1281 ms | 0.3% ↓ | 3.4% ↓ | 54.8% ↓ |
| tradebeans | 10033 ms | 13.2% ↑ | 1.7% ↓ | 1.1% ↑ |
| tradesoap | 9334 ms | 4.5% ↑ | 1.2% ↑ | 11.8% ↓ |

Table 4: Comparision of VMs: (A) OpenJDK; (B) OpenJDK with NMT enabled; (C) Oracle HotSpot; and (D) our sanitizing OpenJDK HotSpot JVM benchmark results. A ↓ indicates a performance decrease.

ond penalty is worth the added security.

In addition to using the DaCapo benchmark suite, we assess the performance of our modifications using the Apache and Socket TLS Clients. This benchmarks uses two JVMS: the unmodified OpenJDK and the modified OpenJDK JVM. We measure performance in terms of request throughput for each client (request per second averaged over four experiments). Figure 6 shows how we controlled for the benchmark varying heap size and the number of requests sent. For each profile, we use a similar number of requests to ensure the number of allocations failures, which lead to GC, are similar. Figure 7 shows the results of this benchmark. The unmodified OpenJDK HotSpot JVM performed better than the modified OpenJDK HotSpot JVM. The performance difference for the Apache TLS Client is 5% for all heap sizes. The Socket TLS Client's impact is 8.25% when the heap size is greater than 4 GiB. Figure 7 shows some anomalous activity for the 4 GiB memory profile for the unmodified JVM, and we are not sure why. We performed several additional experiments, and we analysed the number of requests across all of the benchmarks, but the results are all consistent with the findings in Figure 7.

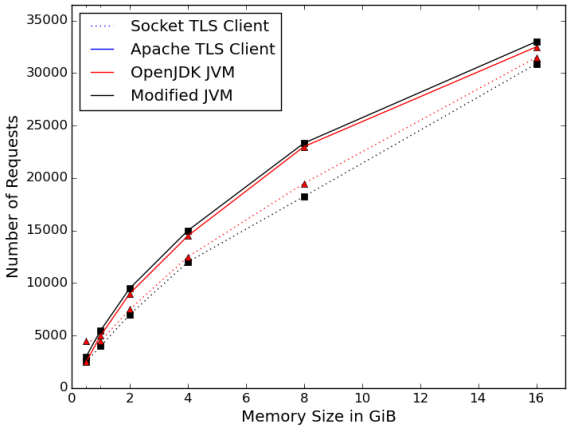
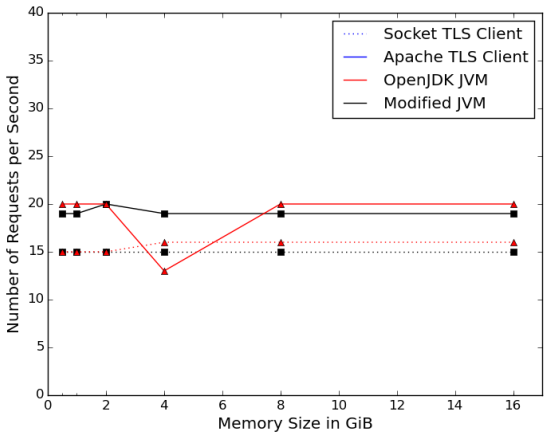



Figure 6: This figure shows the average number of requests for each of the Socket and Apache TLS Clients configurations.

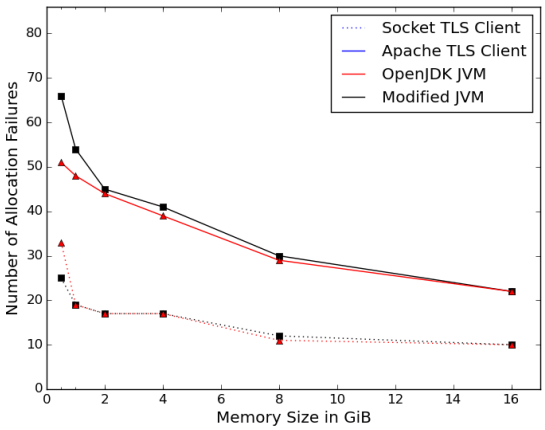
Our conclusions from these benchmarks are that a 50% penalty may be possible with a memory or data intensive Java program. We would not recommend using our modified JVM in these cases; it would be advisable to use a database that stores data encrypted and provides methods for manipulating or handling the data in a secure manner. For more general purpose applications and services like `tradesoap` or our own synthetic web client, the trade-off of performance for security are reasonably palatable.

## 6 Discussion and Future Work

Cleansing latent secrets from managed memory is a challenging problem, and application or runtime demands are going to dictate how these challenges are addressed. Garbage collectors manage the Java heap, and these tasks typically operate when an allocation failure occurs. Parallel collectors might offer some relief to these issues, since data sanitation can be spread across more than one thread, but a modified heap structure and garbage collection implementation are needed to address sanitation issues in a efficient manner. Of course, each and every garbage collection system must be modified to take data sanitation into account. The only way to implement data sanitation without modifying the JVM would be to deliberately kill and restart the JVM on a regular basis. This might be acceptable in some circumstances, but it's not a general-purpose solution.




(a) Requests per second for the Socket TLS and Apache TLS Clients as the heap size increases.




(b) Allocation failures for the Socket TLS and Apache TLS clients as the heap space and number of requests increase.

Figure 7: These figures in conjunction with Figure 6 show the average request throughput and number allocation failures for the Socket and Apache TLS clients running on the OpenJDK JVM or the modified JVM.

GC operations on the young generation happen frequently, so any modification that slows them down will have a disproportionate impact on the final system performance. In contrast, GC operations on the tenured generation are less frequent, but the tenured generation accumulates objects where they remain until a full garbage collection occurs. If the heap is sufficiently large or the program is not very active, full garbage collections may never happen, leaving latent secrets vulnerable to exposure, requiring additional expense to be spent on the tenured generations. We simulate this by manually invoking the garbage collector, but a production-quality garbage collector would need to have sanitation designed into it from the beginning.

One possible future strategy might be to explicitly segregate sensitive data into its own *monitored space* that identifies and removes latent secrets promptly. Developers need the ability to define data lifetimes [4] at a high-level from within Java, or any other managed language. Java currently has meta-data tags, known as *annotations*, that help with the compile-, build-, and runtime operations. *Data lifetime* annotations could help the JVM handle, store, and sanitize these data items without causing a performance burden on the rest of the JVM.

Figure 8 shows a hypothetical generational heap with an additional monitored space. In the monitored space, memory might be explicitly reference counted, allowing for immediate sanitation when an object dies. Furthermore, these objects could have explicit "destroy" APIs, so applications can explicitly kill them, as well as data lifetime annotations an executive task that deletes them when they're expired.

Functionally, references from the main heap to the monitored space would act like *weak references*, making it clear to the application author that sensitive, monitored data could disappear at any time, and must be checked explicitly before each use.

Such a strategy sounds straightforward, but it will have a variety of thorny problems. Consider, for example, all the layers of a protocol stack. Sensitive data can touch on many of these layers. Buffers are assembled and passed along, copies being made all the way. Zero-copy IO techniques (e.g., IO-Lite [16]) could help with this, but that requires the entire stack to be engineered around the particular buffer management strategy. In other words, a change like this will break existing APIs and will require careful engineering of libraries.

This suggests that a customized garbage collector design might be preferable. The design of this is future work, but even our modified version of the HotSpot GC shows that it's entirely possible to achieve reasonable performance on many (but not all) workloads, while gaining significant reductions in the volume of latent secrets.

## 7 Conclusion

Java and the HotSpot JVM are going to be around for decades to come. This runtime offers a rich set of development tools and libraries that help engineers construct and deploy useful software. However, servers and services are susceptible to a number of attacks through a variety of vectors, so there is no guarantee that the system

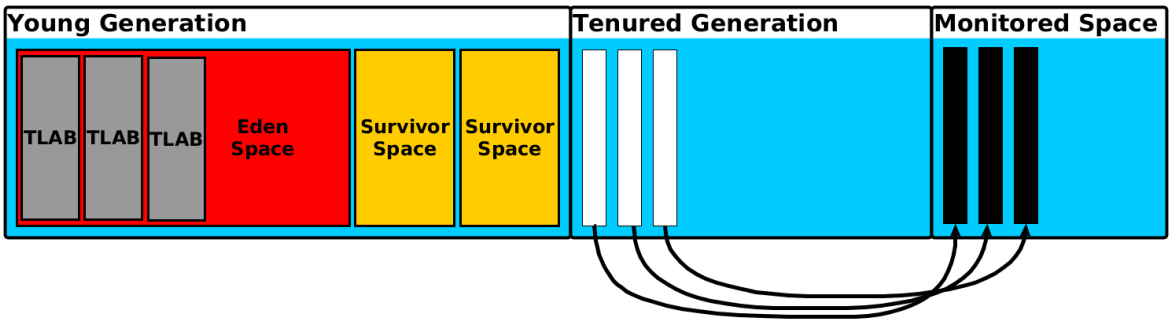



Figure 8: A generational heap layout that uses a monitored space where *sensitive* data is stored and maintained.

where the software executes will remain free from compromise. Attackers evolve quickly, and they will realize that the JVM does not effectively sanitize internal or Java heap memory. This lack of sanitation can compromise sensitive data and lead to unforeseen impacts and consequences. We have taken a proactive approach to this problem by measuring its existence and developing several strategies to help mitigate the problem.

Problems with managed runtime environments like Java and the JVM are well known, but they are not well understood. Our research provides several fundamental elements. We establish the heaps capacity to retain latent secrets. Furthermore, we show that as heaps increase in size the number of latent secrets also increases. Cryptographic libraries should be protecting sensitive data such as keys, but we find that Oracle's JCE implementation of TLS 1.2 does not attempt to eliminate key data.

Given the lack of sanitation in the Java heap, we demonstrate several approaches that reduce the accumulation of sensitive data. When we coordinate these techniques, the number of TLS keys are reduced dramatically. To accomplish this feat, we first modify the JVM to zero unused heap space in the young generation. Second, the tenured generation is also wiped when the dead objects or live objects are encountered during the mark-sweep-compact collection algorithm. We also zero unused heap space after the garbage collection in this space. Finally, we show that four iterations of garbage collection with a time interval spacing of 30 milliseconds are required to completely clear the Java heap, when all activity ceases. These four iterations are necessary, because the tenured space will not otherwise be collected.

We define how to improve performance of garbage collection implementations while keeping data security in mind. Our proposed design modifies the overall structure of the heap, carving out a segment specifically for sensitive data. The design also exploits Java annotations, which can be used to inform the runtime about how to properly handle specific types of data. This design keeps execution and runtime efficiency in mind while allowing for the timely and effective sanitation of data.

## 8 Acknowledgments

Special thanks to the The Cover of Night, LLC for providing the hardware and infrastructure necessary for performing our experiments. We would like to thank Wireshark for accepting our patches to help improve TLS 1.2 decryption when we supplied the premaster secrets necessary to decrypt the session. This research was supported in part by NSF grants CNS-1409401 and CNS-1314492 and the National Physical Science Consortium Fellowship.

## References

[1] BLACKBURN, S. M., GARNER, R., HOFFMANN, C., KHANG, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., ET AL. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices* (2006), vol. 41, ACM, pp. 169–190.

[2] CASE, A. Memory analysis of the dalvik (android) virtual machine. In *Source Seattle* (Dec. 2011).

[3] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 22–22.

[4] CHOW, J., PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Berkeley, CA, USA, 2005), SSYM'05, USENIX Association, pp. 22–22.

[5] COHEN, J. Garbage collection of linked data structures. *ACM Comput. Surv. 13*, 3 (Sept. 1981), 341–367.

[6] COHEN, M. Rekall memory forensics framework. In *DFIR Prague 2014* (2014), SANS DFIR.

[7] DSILVA, V., PAYER, M., AND SONG, D. The correctness-security gap in compiler optimization. In *IEEE CS Security and Privacy Workshop* (San Jose, CA, 2015).

[8] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J.,

APPELBAUM, J., AND FELTEN, E. W. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM 52*, 5 (May 2009), 91–98.

[9] HARRISON, K., AND XU, S. Protecting cryptographic keys from memory disclosure attacks. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on* (2007), IEEE, pp. 137–143.

[10] HILGERS, C., MACHT, H., MÜLLER, T., AND SPREITZENBARTH, M. Post-mortem memory analysis of cold-booted android devices. In *Proceedings of the 2014 Eighth International Conference on IT Security Incident Management & IT Forensics* (Washington, DC, USA, 2014), IMF '14, IEEE Computer Society, pp. 62–75.

[11] JIN, W., COHEN, C., GENNARI, J., HINES, C., CHAKI, S., GURFINKEL, A., HAVRILLA, J., AND NARASIMHAN, P. Recovering c++ objects from binaries using inter-procedural data-flow analysis. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014* (2014), ACM, p. 1.

[12] JONES, R., HOSKING, A., AND MOSS, E. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 2011.

[13] JONES, R. E., AND RYDER, C. A study of java object demographics. In *Proceedings of the 7th international symposium on Memory management* (2008), ACM, pp. 121–130.

[14] MÜLLER, T., AND SPREITZENBARTH, M. Frost: Forensic recovery of scrambled telephones. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security* (Berlin, Heidelberg, 2013), ACNS'13, Springer-Verlag, pp. 373–388.

[15] ORACLE. Java platform, standard edition troubleshooting guide, 2015.

[16] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Io-lite: A unified i/o buffering and caching system. In *Third Symposium on Operating Systems Design and Implementation (OSDI'99)* (New Orleans, LA, Feb. 1999).

[17] STTTGEN, J., AND COHEN, M. Robust linux memory acquisition with minimal target impact. *Digital Investigation 11, Supplement 1* (2014), S112 – S119. Proceedings of the First Annual DFRWS Europe.

[18] SUN MICROSYSTEMS. Memory management in the java hotspottm virtual machine, Apr. 2006.

[19] VIEGA, J. Protecting sensitive data in memory, 2001.

[20] VÖMEL, S., AND FREILING, F. C. A survey of main memory acquisition and analysis techniques for the windows operating system. *Digit. Investig. 8*, 1 (July 2011), 3–22.