# Towards Protecting Sensitive Data in Java Virtual Machine

Lin Deng[1], Matt Benke[2], Tyler Howard[1], Matt Krause[1], Aman Patel[1], Montrell Jubilee[1], and Dalton Watts[1]
[1]Department of Computer and Information Sciences
Towson University, Maryland, USA
LDeng@towson.edu, {thowar4,mkraus9,apatel23,mjubil1,dwatts3}@students.towson.edu
[2]National Security Agency

***Abstract*— All kinds of Java-based applications are widely used by government agencies and financial institutions. Every day, people's sensitive data, such as credit card numbers and passwords, are frequently processed by these applications. Research has found that Java Virtual Machine (JVM), an essential component for executing Java-based applications, stores data in memory for an unknown amount of time even after the data are no longer used. This mismanagement puts all the data, sensitive or non-sensitive, in danger and raises a huge concern to all the Java-based applications globally. This problem has serious implications for many "secure" applications that employ Java-based frameworks or libraries with a severe security risk of having sensitive data that can be accessed by attackers after the data are thought to be cleared. This paper presents a prototype of a secure Java API we design through an undergraduate student research project. The API is implemented using direct Byte buffer so that sensitive data are not managed by JVM garbage collection. We also implement the API using obfuscation so that data are encrypted. Using an initial experimental evaluation, the proposed secure API can successfully protect sensitive data from being accessed by attackers.**

***Keywords—Cybersecurity, Java, Java Virtual Machine, Garbage Collection***

## I. INTRODUCTION

Due to the significant growth of mobile and web applications, Java, as a fundamental programming language used by these applications, becomes dominant in the community of program developers. Statistics show that Java is the most popular and most in-demand programming language [1]. Developers understand the importance of keeping their programs secure. Generally, Java is considered as a safe language with decent security features. Meanwhile, attackers start to gain access to more sophisticated hacking tools that increase the number of cyber-related threats and cause more security vulnerabilities. Consequently, preventing potential cyber-attacks and maintaining confidentiality, integrity, and availability of cyberspace become a top priority task for IT practitioners, researchers, and government agencies.

Java Virtual Machine (JVM) is a key component of the Java Runtime Environment (JRE). JVM is necessary for Java programs to execute. JVM can be considered as a virtual computing machine that enables Java programs to be platform independent, i.e., Java programs can use the same interfaces and libraries, and then, different JVM implementations for different operating systems can accommodate the running environment by bridging JVM instructions of Java programs and local operating systems [2]. In JVM, the heap is the data area in memory that is used to store Java instances, arrays, etc. Heap is created when JVM starts to run a Java program. Since it is in the system memory along with the running of the program, its size is limited. Therefore, JVM uses the garbage collection (GC) mechanism to deallocate those objects that are no longer in use and manage the fragmentation of the heap.

However, research finds that JVM stores data in the heap for an unknown amount of time after the data has been used [3][4]. If these data include sensitive information, such as our passwords or credit card numbers, they can be a vulnerable target of cyber-attacks. This problem is inherently getting worse because GC is outside the developer's control in JVM. The exposure of sensitive data impacts all kinds of web and mobile apps developed using Java. Given the popularity of Java, having a security vulnerability on the technology of this scale can cause severe problems. Research into the potential solutions to this vulnerability is necessary and imperative. To address this problem, in this paper, we propose our initial design and implementation of a secure API for protecting sensitive data in JVM. We first investigate how the inherent property of JVM, i.e., GC, impacts the security of sensitive data in Java programs. Based on the investigation, then, we implement a secure API that can protect sensitive data. Finally, we use an initial evaluation to assess the effectiveness of the secure API for protecting sensitive data from leaking to attackers.

The rest of this paper is organized as follows: Section II goes through the background of JVM and GC; Section III describes the design and implementation of proposed secure API; Section IV gives an overview of related research, and the paper concludes and suggests future work in Section V.

## II. BACKGROUND

### A. Java Virtual Machine (JVM) and Garbage Collection(GC)

Unlike other programming languages, such as C, that provide programmers the freedom to manually deallocate memory, JVM uses a GC mechanism to automatically deallocate objects no longer used by programs in the heap [5]. In this way, programmers do not need to pay attention to the memory management while writing Java programs. Specifically, when an object is created by the *new* keyword, JVM allocates the memory space for the object on the heap at runtime. If the object is no longer referenced by the program, the GC can

recycle the heap space occupied by the object, so that other new objects can use the space. In this way, programmers can save time and effort during software development. Also, since GC limits the ability of programmers to manipulate the memory space, either accidentally or purposely, it helps reducing the risk of having security vulnerabilities and software defects.

Another major feature in GC mechanism is to manage the heap fragmentation. Usually, during the execution of Java programs, objects are frequently created and unreferenced, which means that their heap space also gets changed all the time. Inevitably, unused memory space may be present between blocks occupied by live objects. The GC can move the locations of these live objects in the heap to consolidate the fragmentation. Note that there are different JVM implementations, such as HotSpot [6] and Eclipse OpenJ9 [7]. They may have different GC mechanisms defined and implemented.

### B. *Disadvantages and Challenges of GC*

Because programmers are not involved in the GC, the GC in JVM must automatically determine which objects are no longer referenced by the program and free the heap space of such objects. Inevitably, this mechanism might have some disadvantages and challenges.

One disadvantage of Java GC is the impact on performance, because the JVM must track every object created by the program and manage their memory space. Researchers have found that JVM GC costs significant overhead during finding unreferenced objects and recycling memory used by them, and proposed innovative techniques to reduce this overhead [8][9][10]. The overhead issue of JVM GC is not the focus of this paper.

Another major concern that associates with GC in JVM is regarding security. As Java-based systems are widely used by government agencies and financial institutions, the more we rely on these systems, the more frequent our sensitive data get processed. Researchers have found security vulnerabilities that target Java objects, which indicates that JVM GC is insufficient to prevent unauthorized access to sensitive data in memory. More specifically, when an object that contains sensitive data becomes unreferenced by the program and ready to be recycled, JVM GC may not act immediately to clear the memory space used by the object, depending on the specific GC mechanisms. Attackers may be able to obtain these sensitive data in the memory, before JVM GC erase them. The situation gets more complicated for String objects because they are immutable. The worst case is that, when a String object gets its reference updated many times during the execution, its old referenced values could still be present everywhere in the heap before the GC assigns the heap space to other objects. Similarly, when the GC rearranges the heap space for defragmentation, some sensitive data are moved to a new place, but the data at the old location may not be wiped out, even though they are no longer referenced. As more and more sophisticated malware and hacking tools are available, it is not very difficult for attackers to obtain these sensitive data in the heap.

## III. Design and Implementation

This section introduces the architectural design and the technical implementation of our proposed technique for securing sensitive data in JVM. Our major goal is to provide a secure Java API that can prevent attackers from accessing or obtaining data in JVM during and after the execution of programs. As an undergraduate student research project, we would like to provide a solution that does not directly modify JVM, as that would be beyond students' technical knowledge and experience. Instead, we want to create an API that could still clear an object's content whenever necessary and should work effectively, efficiently, and securely. Therefore, our proposed API includes two major design aspects: (i) It uses a direct byte buffer that outside the heap; (ii) It uses obfuscation to encrypt the data.

First, we aim to tackle the challenge that GC moves data around in the heap. A solution is to store sensitive data outside the heap space controlled by the GC, so that the data cannot be relocated anymore. Inspired by an online post [11], we design our secure API with a direct *ByteBuffer* from *java.nio* package. As shown in Figure 1, direct byte buffers reside outside of the normal garbage-collected heap of JVM and, thus, are outside the control of GC. Sensitive data stored in direct byte buffer will not be moved or copied around in the memory. Regarding the performance aspect, we hypothesize that using direct byte buffers is faster than usual processing through the heap in JVM, since direct byte buffers have direct connections with I/O at the OS level. However, the actual performance may vary depending on different JVM implementations, different OS platform, and specific Java programs. In the initial phase of the project, we have not experimentally compared the two approaches.

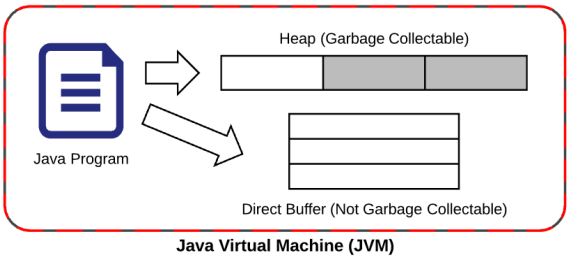


Figure 1: Direct Buffer in JVM

Second, our secure API uses obfuscation to protect sensitive data so that attackers cannot read the data from a memory dump, for the limited amount of time the data exist in memory. Obfuscation is widely used in the domains of software and cybersecurity to make information or data hard to read or understand by others. As an initial implementation of our secure API, we use an XOR cipher as the method of obfuscation. The API allows the appending of individual characters to a buffer that is then converted into bytes and XORed with a key to obfuscate the data. This method is simple and inexpensive. Meanwhile, our secure API avoids obfuscating or deobfuscating all chars at once. Otherwise, attackers may be able to recover the data from memory. Similarly, our API disables *toString()* and *clone()* methods to ensure the data security.

Figure 2 illustrates an overview of the design of our API. The *SecureData* class includes *SecureCharBuffer* as an attribute of the secure buffer data. The API supports the explicit clearing of data once the direct byte buffer is closed, by calling the *close()* method. Then, the entire object is immediately removed from memory, since the data exist at only one fixed location in the memory. Similarly, while the buffer could be cleared by calling

the *close()* method, we also implemented an useful interface called *AutoCloseable* that can close the buffer automatically if the programmer wrapped the operations inside a try-catch. These simply minimized reliance on the programmer to perform certain actions correctly like remembering to close the buffer and made the API easy to use. From the perspective of functionality, our secure API not only supports String objects, but also can be used towards other custom defined data types. Since data of Java primitive types are stored in the stack, instead of the heap, our initial design and implementation do not consider these primitive types.

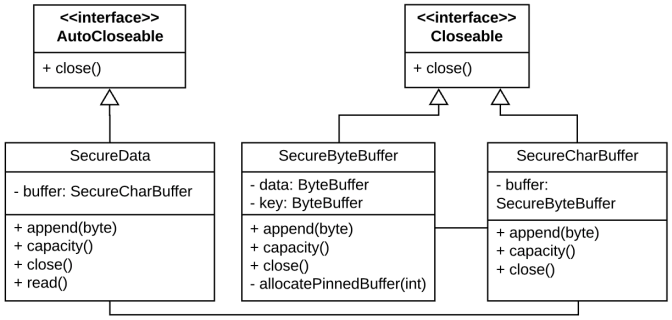


Figure 2: Design Overview

After implementing the API, we use Visual VM to obtain heap dumps, and check for the sensitive data we deliberately process at various times throughout runtime. We can successfully confirm that no sensitive data exist in memory.

## IV. Related Work

Researchers designed different approaches to protect data in JVM. Liu et al. proposed a programming mechanism, called ObEx [12], that privately stores confidential information and allows clients to use predefined mathematical operations upon them. Similarly, these objects can have a specified lifetime and access limit, and cannot be copied as they are simply singleton objects. Sampaio and Garcia [13] went into details concerning secure programming tactics that allow for developers to continuously detect new security vulnerabilities early, while writing new lines of code. Their prototype is called Early Security Vulnerability Detector (ESVD). Kuleshov [14] outlines a simple Java API with the ability to modify bytecode and illustrates common patterns that can be implemented in order to achieve specific transformations.

## V. Conclusion and Future Work

In cybersecurity, data leakage is a severe and significant problem that impacts every individual's life. For Java-based applications, research finds that GC in JVM cannot appropriately erase sensitive data after they are unreferenced. This paper introduces our study that aims to protect sensitive data in JVM. We design and implement a secure API that can encrypt sensitive data and store the data outside the heap controlled by GC, so that attackers cannot access the data.

For future work, we plan to continue improving the API by adding more secured encryption mechanisms, such as AES or DES. We would also like to evaluate the performance of the API by comparing it to other similar techniques.

## Acknowledgment

This undergraduate student research project is supported via the Information Security Research and Education (INSuRE) project [15].

## References

[1] TIOBE Software BV, "TIOBE Index | TIOBE - The Software Quality Company," *Tiobe*, 2018. [Online]. Available: https://www.tiobe.com/tiobe-index/. [Accessed: 16-Aug-2018].

[2] B. Venners, *Inside the Java Virtual Machine*. McGraw-Hill, 1997.

[3] A. Pridgen, S. L. Garfinkel, and D. S. Wallach, "Present but Unreachable: Reducing Persistentlatent Secrets in HotSpot JVM," *Proc. 50th Hawaii Int. Conf. Syst. Sci.*, 2017.

[4] Charlie Gracie, "JavaOne 2016 - JVM assisted sensitive data," *JavaOne 2016*. [Online]. Available: https://www.slideshare.net/CharlieGracie/javaone-2016-jvm-assisted-sensitive-data. [Accessed: 07-Feb-2018].

[5] R. Jones, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[6] "HotSpot Group." [Online]. Available: http://openjdk.java.net/groups/hotspot/. [Accessed: 16-Aug-2018].

[7] "OpenJ9." [Online]. Available: http://www.eclipse.org/openj9/. [Accessed: 16-Aug-2018].

[8] Y. Zhang, L. Yuan, T. Wu, W. Peng, and Q. Li, "Just-in-Time Compiler Assisted Object Reclamation and Space Reuse," vol. LNCS-6289, Springer, 2010, pp. 18–34.

[9] D. Bacon, P. Cheng, and V. T. Rajan, "The Metronome: A simpler approach to garbage collection in real-time systems," *Move to Meaningful Internet Syst. ...*, vol. 2889, pp. 466–478, 2003.

[10] S. M. Blackburn and K. S. McKinley, "Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*, 2008, vol. 43, no. 6, p. 22.

[11] ωєѕтση, "Protecting strings in JVM Memory – ωєѕтση – Medium." [Online]. Available: https://medium.com/@_west_on/protecting-strings-in-jvm-memory-84c365f8f01c. [Accessed: 16-Aug-2018].

[12] Y. Liu, Z. Song, and E. Tilevich, "Querying Invisible Objects: Supporting Data-Driven, Privacy-Preserving Distributed Applications," in *Proceedings of the 14th International Conference on Managed Languages and Runtimes - ManLang 2017*, 2017, pp. 60–72.

[13] L. Sampaio and A. Garcia, "Exploring context-sensitive data flow analysis for early vulnerability detection," *J. Syst. Softw.*, vol. 113, pp. 337–361, Mar. 2016.

[14] E. Kuleshov, "Using the ASM Framework to Implement Common Java Bytecode Transformation Patterns," *Proc. AOSD - Intl. Conf. Asp. Softw. Dev.*, pp. 1–7, 2007.

[15] A. Sherman, M. Dark, A. Chan, R. Chong, T. Morris, L. Oliva, J. Springer, B. Thuraisingham, C. Vatcher, R. Verma, and S. Wetzel, "INSuRE: Collaborating Centers of Academic Excellence Engage Students in Cybersecurity Research," *IEEE Secur. Priv.*, vol. 15, no. 4, pp. 72–78, 2017.