



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

IMPROVING THE SECURITY OF GARBAGE COLLECTORS IN THE JAVA VIRTUAL MACHINE

ZLEPŠOVÁNÍ BEZPEČNOSTI GARBAGE COLLECTORŮ V JAVA VIRTUAL MACHINE

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR

AUTOR PRÁCE

Bc. PATRIK ČERBÁK

SUPERVISOR

VEDOUCÍ PRÁCE

Ing. DAVID KOZÁK, Ph.D.

BRNO 2026

Master's Thesis Assignment



172463

Institut: Department of Intelligent Systems (DITS)
Student: **Čerbák Patrik, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Cybersecurity
Title: **Improving the Security of Garbage Collectors in the Java Virtual Machine**
Category: Compiler Construction
Academic year: 2025/26

Assignment:

1. Familiarize yourself with garbage collectors and their implementation techniques, focusing on garbage collectors in the Java Virtual Machine.
2. Next, familiarize yourself with attack techniques that exploit unauthorized access to protected data. Focus on methods based on garbage collector deficiencies.
3. Select at least one garbage collector from the Java Virtual Machine, and propose how to introduce preventive measures against such attacks. Implement the proposed measures as a modification of one of the Java Virtual Machine implementations.
4. Evaluate the modified version of Java Virtual Machine on a representative set of benchmarks.

Literature:

1. Jones, Richard, Antony Hosking, and Eliot Moss. The garbage collection handbook: the art of automatic memory management. Chapman and Hall/CRC, 2023.
2. Maxim Anikeev, Felix C. Freiling, Johannes Götzfried, Tilo Müller, Secure garbage collection: Preventing malicious data harvesting from deallocated Java objects inside the Dalvik VM, Journal of Information Security and Applications, Volume 22, 2015, Pages 81-86, ISSN 2214-2126, <https://doi.org/10.1016/j.jisa.2014.10.001>.
3. Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012). Association for Computing Machinery, New York, NY, USA, 133–143. <https://doi.org/10.1145/2338965.2336769>
4. Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC'12). USENIX Association, USA, 28.
5. Kostya Serebryany, Chris Kennelly, Mitch Phillips, Matt Denton, Marco Elver, Alexander Potapenko, Matt Morehouse, Vlad Tsyrklevich, Christian Holler, Julian Lettner, David Kilzer, and Lander Brandt. 2024. GWP-ASan: Sampling-Based Detection of Memory-Safety Bugs in Production. In Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24). Association for Computing Machinery, New York, NY, USA, 168–177. <https://doi.org/10.1145/3639477.3640328>
6. <https://docs.oracle.com/en/java/javase/17/gctuning/garbage-first-g1-garbage-collector1.html>

Requirements for the semestral defence:

The first two points of the assignment and at least some initial work on the third point.

Detailed formal requirements can be found at <https://www.fit.vut.cz/study/theses/>

Supervisor: **Kozák David, Ing.**
Head of Department: Kočí Radek, Ing., Ph.D.
Beginning of work: 1.11.2025
Submission deadline: 20.5.2026

Approval date: 3.11.2025

Abstract

This thesis focuses on improving the security of the Garbage-First garbage collector (G1 GC) in the Java Virtual Machine (JVM). The increase in security is achieved through continuous relocation of memory regions used by the garbage collector and by “locking” the original memory regions in which they were located. Thanks to this relocation, the dereference of dangling pointers (or other pointers created, for example, by a buffer overflow), which are still incorrectly pointing to the original memory, would be detected. This makes it much easier to detect and debug memory vulnerabilities in this garbage collector.

This extension of G1 GC is called *SanitizeGC*, and it was tested with a set of custom tests as well as with a G1 test suite already present in the JVM’s code. Apart from that, its functionality was checked on a few real-world applications. There were also two special scenarios created that introduce bugs on purpose to showcase that the extension can catch them efficiently (which was successful), at the cost of worse performance (4.5 times slower on average in high memory workload scenarios) and higher physical memory usage (1.1 times more on average, doubled in the worst case). Giving evidence that *SanitizeGC* is a useful diagnostic tool in the development and testing phases of JVM with acceptable overhead.

Abstrakt

Tato diplomová práce se zaměřuje na zlepšení bezpečnosti Garbage-First garbage collectoru (G1 GC) v Java Virtual Machine (JVM). Zvýšení bezpečnosti je dosaženo průběžnou relokací paměťových regionů používaných tímto garbage collectorem a “uzamčením” původní paměti, ve které se nacházely. Díky této relokaci je možné odhalit dereferenci “dangling” ukazatelů (nebo jiných ukazatelů vzniklých například přetečením bufferu), které stále chybně odkazují na původní paměť. To výrazně usnadňuje detekci a ladění paměťových zranitelností v tomto garbage collectoru.

Toto rozšíření G1 GC se nazývá *SanitizeGC* a bylo testováno jak pomocí sady vlastních testů, tak s využitím testovací sady pro G1, která již byla součástí kódu JVM. Kromě toho byla jeho funkcionality ověřena na několika reálných aplikacích. Vytvořeny byly také dva speciální scénáře, které záměrně zavádějí chyby, aby se ukázalo, že je toto rozšíření dokáže efektivně zachytit (což bylo úspěšné), i když za cenu nižšího výkonu (v průměru je 4,5krát pomalejší při vysokém zatížení paměti) a větší spotřeby fyzické paměti (v průměru 1,1krát vyšší, v nejhorsím případě dvojnásobná). To poskytuje důkaz, že *SanitizeGC* je užitečným diagnostickým nástrojem ve fázích vývoje a testování JVM s přijatelnou režijí.

Keywords

Java, OpenJDK, JVM, HotSpot, garbage collectors, G1 GC, security, dangling pointers, use-after-free vulnerability

Klíčová slova

Java, OpenJDK, JVM, HotSpot, garbage collectors, G1 GC, bezpečnost, dangling ukazatele, use-after-free zranitelnost

Reference

ČERBÁK, Patrik. *Improving the Security of Garbage Collectors in the Java Virtual Machine*. Brno, 2026. Master’s thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. David Kozák, Ph.D.

Rozšířený abstrakt

Tato diplomová práce se zabývá problematikou bezpečnosti garbage collectorů v Java Virtual Machine (JVM), konkrétně se zaměřuje na vylepšení odolnosti Garbage-First garbage collectoru (G1 GC) proti softwarovým zranitelnostem způsobeným špatnou správou paměti. Hlavní motivací projektu je skutečnost, že ačkoliv je Java považována za paměťově bezpečný jazyk, samotná implementace jejího běhového prostředí v jazyce C++ zůstává náchylná k chybám typu use-after-free, double-free nebo buffer overflow.

Tyto chyby jsou v rozsáhlých systémech, jakým je OpenJDK, velmi obtížně detekovatelné a laditelné, protože se jejich následky často projeví až s velkým časovým odstupem od jejich samotného vzniku. Práce proto představuje rozšíření nazvané *SanitizeGC*, které implementuje mechanismus průběžného přemísťování paměťových regionů a jejich následného uzamykání s cílem zachytit neoprávněný přístup k paměti prostřednictvím “dangling” nebo jiných neplatných ukazatelů.

Základ navrženého řešení spočívá v modifikaci správy paměťových regionů, které G1 GC využívá k ukládání objektů. V běžném režimu G1 GC rozděluje svou předalokovanou paměť na stejně velké bloky (regiony), které jsou přiřazovány jednotlivým generacím objektů. *SanitizeGC* do tohoto procesu zasahuje v momentu, kdy je region vyjmut ze seznamu volných regionů pro novou alokaci. V tomto kroku *SanitizeGC* alokuje pro daný region zcela novou adresu v rámci virtuálního adresního prostoru a původní umístění regionu “zamkne”.

Pro uzamčení původní paměti se využívá protekce paměti, která se nastaví na *NONE*, což způsobí, že jakýkoliv pokus o čtení nebo zápis na tuto adresu vyvolá okamžitý pád aplikace následovaný výpisem, kde k tomuto pádu došlo. Díky tomu lze identifikovat chyby v logice garbage collectoru nebo v nativním kódu, které by jinak mohly vést k tiché korupci dat nebo bezpečnostním únikům.

Klíčovou věcí při implementaci *SanitizeGC* byla nutnost zachovat kompatibilitu s vnitřními mechanismy G1 GC, které spoléhají na statické výpočty s adresami, například u takzvaných “card tables” (datová struktura rozděluje paměť na malé fixní bloky, které slouží k efektivnímu sledování referencí mezi jednotlivými regiony bez nutnosti prohledávat celou paměť). Protože se adresy regionů neustále mění, bylo nutné vytvořit mapovací komponentu, která slouží k obousměrnému překladu mezi novou, reálně používanou adresou a původní adresou v logickém adresním prostoru.

Tato komponenta využívá hašovací mapy a k identifikaci regionu (získání klíče) používá bitové posuny podle velikosti regionů. Při každém přístupu ke strukturám, jako jsou právě card tables, nebo při výpočtech rozdílu ukazatelů, musí dojít k přemapování adresy, aby vnitřní logika JVM pracovala správně. Celkově bylo v kódu HotSpotu identifikováno a upraveno více než třicet míst vyžadujících tento překlad, což zahrnovalo i zásahy do kódu, který generuje strojový kód pro architekturu x86-64.

Z hlediska efektivity nakládání s pamětí využívá *SanitizeGC* optimalizační mechanismus anonymní paměti v jádře Linuxu. Při vracení regionu do seznamu volných bloků dochází k uvolnění fyzických stránek, zatímco virtuální adresy zůstávají zamknuty. Tento přístup minimalizuje plýtvání fyzické paměti, jediný zatížený prostor je ten virtuální (který je však v moderních systémech dostatečně velký). *SanitizeGC* je v současné fázi zaměřen na platformu Linux x86-64 a podporuje verzi Java 23, přičemž jsou v implementaci prozatím ignorovány takzvané humongous regiony (regiony, které zabírají prostor několika normálních regionů) a je vyžadováno vypnutí komprese ukazatelů (provede se automaticky), což výrazně zjednodušuje přemapování adres.

Praktická část práce zahrnuje rozsáhlé testování a měření výkonu na sadě dvanácti zatěžových benchmarků i reálných aplikacích, jako jsou SQL databáze H2 nebo frame-

work Quarkus (pro ověření běžné funkcionality). Výsledky ukazují, že zvýšená úroveň zabezpečení je vykoupena nezanedbatelnou režií. SanitizeGC je v benchmarkích průměrně 4,5 krát pomalejší než standardní JVM. Spotřeba paměti vzrostla průměrně o 10 procent, v extrémních případech však byla až dvojnásobná oproti běžnému stavu. Tyto výsledky naznačují, že by mohl být tento nástroj využitelný zejména ve fázi vývoje a testování samotného JVM (nebo některých uživatelských aplikací).

Nejdůležitějším důkazem funkčnosti SanitizeGC je úspěšná detekce záměrně vložených chyb v rámci takzvaného bug seedingu. V prvním scénáři byla do zdrojového kódu JVM vložena chyba simulující opomenutí programátora při práci s takzvanými “handle” strukturami, které slouží pro bezpečné uchování ukazatele v případech, kdy je objekt, na který ukazují, přesunut v důsledku kolekce paměti. V tomto bug seedingu byl použit surový ukazatel na objekt namísto bezpečného odkazu přes handle a byla vyvolána kolekce paměti, která tento objekt přesunula (ze surového ukazatele se tedy stal dangling ukazatel). Standardní JVM tento chybný přístup k paměti nezachytilo a pokračovalo v běhu s neplatnými daty, zatímco SanitizeGC pád aplikace vyvolalo okamžitě v momentu pokusu o zápis na zamčenou původní adresu objektu.

Druhý scénář prokázal schopnost nástroje odhalit chyby v uživatelském kódu využívajícím Java Native Interface (JNI). Šlo o situaci, kdy nativní kód v jazyce C přistupoval k prvkům pole nesprávně i mimo kritickou sekci, ve které má přístup povolený. SanitizeGC tento neoprávněný přístup detekovalo po proběhnutí první kolekce paměti, která pole přesunula. Tímto SanitizeGC potvrdilo svou užitečnost i pro koncové vývojáře pracující s JNI nebo s jinými “paměťově nebezpečnými” mechanismy Javy.

Hlavním cílem práce bylo vytvořit “proof of concept”, který by jasně ukazoval reálnou použitelnost a zlepšení bezpečnosti v JVM. Toto bylo splněno, vytvořený proof of concept představuje funkční základ pro další práci v rámci tohoto tématu. Budoucí vývoj by se měl zaměřit na podporu humongous regionů, optimalizaci pro snížení výkonové penalizace a přesun na novější verze Java. V plánu je také sdílení těchto výsledků v rámci širší OpenJDK komunity s případným samotným cílem přidat toto vylepšení do OpenJDK.

SanitizeGC může sloužit jako diagnostický nástroj, který transformuje tiché a nebezpečné chyby správy paměti na jasně identifikovatelné pády, což výrazně zvyšuje šanci na jejich včasné odstranění.

Improving the Security of Garbage Collectors in the Java Virtual Machine

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. David Kozák, Ph.D. The supplementary information was provided by Francisco Ferrari Bihurriet and Martin Balao Alonso. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis. I have used Grammarly and occasionally Gemini to correct spelling and adjust the stylistics of the text.

.....
Patrik Čerbák
May 17, 2026

Acknowledgements

I would like to sincerely thank everyone who supported me during the completion of this thesis. On the work side, I would like to express my appreciation to Francisco Ferrari Bihurriet and Martin Balao Alonso, who have helped me understand the concepts of Java, HotSpot, garbage collectors, and even programming in general. I would also like to thank Thomas Fitzsimmons, Thomas Stüfe, Jiří Vaněk, and Andrew Dinn for their help, consultations, and feedback on the project. On the university side, my thanks go to Ing. David Kozák, Ph.D., mainly for helping me with every official requirement for this thesis and also for his input on the contents and the text of this thesis, as well as for the input on the project itself.

Lastly, I would like to thank my girlfriend Anna, my family, and everyone else I forgot to mention for supporting me during the writing and, more generally, during my studies at FIT.

Contents

1	Introduction	3
2	Garbage Collectors	5
2.1	Basic Java-Related Terms	5
2.2	Garbage Collector	7
2.3	Garbage-First (G1) Garbage Collector	7
3	Potential Memory Vulnerabilities in Garbage Collectors	14
3.1	Dangling Pointers	14
3.2	Use-After-Free Vulnerability	16
3.3	Double-Free Vulnerability	18
3.4	Buffer Overflow Vulnerability	19
4	Planning SanitizeGC Improvements	21
4.1	Inspiration	21
4.2	The Main Idea	22
4.3	Alternative Approach Considered	27
4.4	Summary of Changes to Implement	28
5	The Implementation of SanitizeGC	30
5.1	The Implementation Platform	30
5.2	Moving the Regions	31
5.3	Locking the Regions	33
5.4	Address Conversion	34
5.5	Quality of Life Changes	41
6	Testing the SanitizeGC Changes	43
6.1	Different Build Types	43
6.2	Functional Tests	43
6.3	Bug Seeding	48
7	Conclusion	58
	Bibliography	60
A	Attachment Directory Structure	64
B	Building and Running the SanitizeGC	65
B.1	Building SanitizeGC	65

B.2	Running Test Suites	66
B.3	Running Benchmarks	66
B.4	Running Bug Seeding	67

Chapter 1

Introduction

Currently, the Java programming language is one of the most used programming languages in the world [36]. It is being used in various applications and companies worldwide, and it continues to be actively developed. At this time, a new version of Java is being released every six months. However, with new features added, Java (and its development and runtime environments) needs to be tested for potential security flaws, so the customers can use it without worrying about such issues.

At the time of writing this thesis, I’m employed at the OpenJDK¹ team at IBM², and part of the tasks there is to find and fix these security flaws and contribute the fixes to the OpenJDK code. The idea for this thesis came from my colleagues from the security sub-team, Martin Balao Alonso³ and Francisco Ferrari Bihurriet⁴.

Currently, memory mismanagement-related vulnerabilities are among the most dangerous that can occur in any software [3, 23]. What’s more, those vulnerabilities are very difficult to debug and identify the actual cause, because most of them are detected much later than they occurred during the code’s execution.

This is where this thesis comes in. The aim of this thesis is to try extending the implementation of the Garbage-First garbage collector (G1 GC, [4], the de facto default garbage collector in OpenJDK) and make it detect unauthorized memory accesses by moving the memory that the G1 GC is using to new places and “locking” the old memory throughout a program’s run (so that an access from a dangling pointer is caught early and the Java Virtual Machine stops).

For this thesis, the main aim was to create a usable proof of concept of this idea that can serve as a diagnostic tool used for catching memory vulnerabilities and test it to have a proof that it is worth pursuing further (which might include optimizing it more and creating a JDK Enhancement Proposal⁵ with the ultimate goal of integrating it to the OpenJDK code). The garbage collector extension in this thesis was implemented on a single common platform and made available as an optional switch that can be enabled at runtime.

¹The open-source implementation of Java.

²The team was at Red Hat when the work on the thesis started, but we were moved to IBM in the meantime as a part of strategic realignment.

³<https://martin.uy/blog/about/index.html> (He later left IBM to pursue his career elsewhere.)

⁴<https://github.com/franferrax>

⁵<https://openjdk.org/jeps/1>

The name of this extension we decided on is **SanitizeGC**. The name is inspired by Google’s AddressSanitizer, and it should invoke the main idea of the project – *sanitization* (making something clean and free of bacteria/bugs⁶) of the garbage collector.

The SanitizeGC extension was successfully designed and implemented as a proof-of-concept for the Linux x86-64 platform with Java version 23. To evaluate its effectiveness and overhead, it was tested on a custom SanitizeGC test suite, the existing G1 test suite, and performance benchmarking across twelve stress tests with varying heap sizes was conducted. Additionally, the tool was tested on a few real-world applications to verify its practical usability. Finally, to demonstrate its diagnostic capabilities, two specific “bug seeding” scenarios were created that intentionally introduce dangling pointer bugs into OpenJDK and a specific user-made piece of code.

In both bug seeding scenarios, standard JVM execution failed to detect the vulnerabilities, but SanitizeGC immediately caught the invalid memory accesses, crashed the program securely, and provided accurate stack traces leading to the causes. However, this diagnostic security comes with a measurable performance cost under heavy workloads. Benchmarking revealed that SanitizeGC is, on average, 4.5 times slower than a standard JVM. Physical memory usage increases, on average, 1.1 times, but can double in worst-case scenarios.

Ultimately, this thesis proves that SanitizeGC is a useful diagnostic tool for developers, mainly during the JVM development and testing phases. Future work will focus on debugging a few complex edge-case failures discovered during testing, implementing support for a few Java features that were skipped in this proof-of-concept, and rebasing the extension to a newer version of Java.

The text of this thesis is structured as follows: Chapter 2 describes garbage collector concepts, focusing mainly on the G1 GC and its aspects important to this project (such as the memory layout it uses, the collection cycle, etc.). Then, in Chapter 3, potential memory-related vulnerabilities are described, which are still being found in software even today. Those include the *use-after-free* vulnerability, the *double-free* vulnerability, and the *buffer overflow* vulnerability.

After that, the changes introduced by SanitizeGC are described in theory in Chapter 4. First, there is a mention of Google’s AddressSanitizer, which inspired this project. Next up, the main idea for improving security (moving and locking memory) is described in detail, along with the necessary changes that needed to be made to make SanitizeGC compatible with the existing JVM code.

Then, Chapter 5 presents the actual implementation of those changes. It goes into technical detail about where and how everything was implemented.

Lastly, the testing of SanitizeGC is mentioned in Chapter 6. It outlines standard testing (on the SanitizeGC and G1 test suites), real-world application testing, benchmarking (including plots of the results), and the two bug seeding scenarios in detail.

⁶<https://dictionary.cambridge.org/dictionary/english/sanitization>

Chapter 2

Garbage Collectors

This chapter describes garbage collectors. Special attention is paid to the Garbage-First garbage collector (or G1 GC for short), which is the de facto default garbage collector in Java [12] and the garbage collector that was chosen for this project.

2.1 Basic Java-Related Terms

Before delving deeper into the garbage collectors, some Java-related terms are explained, which are then used throughout the thesis.

2.1.1 Java Programming Language

Java is a high-level, general-purpose, memory-safe, object-oriented programming language that was designed by James Gosling at Sun Microsystems in 1995 [37]. Since then, it has gained significant popularity, particularly in the enterprise sphere, where it remains one of the most widely used programming languages to this day. According to GitHub’s October 2025 data, it is the fourth-most-used programming language on the platform [7].

In 2010, Oracle acquired Sun Microsystems and continues to lead Java development to this day (although it is not the only company developing Java, as the main implementation is now open-source). As of the time of writing this thesis, a new Java version comes about every six months, and it continues to bring new features and improvements to the language. [37]

Java is a multi-platform language, meaning that the same compiled source code can be run on any machine with the Java Virtual Machine installed. That is achieved by Java not compiling sources into platform-specific binaries, but rather into bytecode (a set of instructions that the virtual machine can run). [37]

2.1.2 Java Virtual Machine

Java Virtual Machine (or JVM for short) is the piece of software that actually runs the compiled Java bytecode. It serves as a layer between the language and the operating system it runs on [9]. It takes the bytecode, and either interprets it instruction by instruction at runtime or just-in-time (JIT) compiles it into platform-specific native code, which should make the program run faster in most cases. Figure 2.1 shows process of compilation and interpretation/JIT compilation.

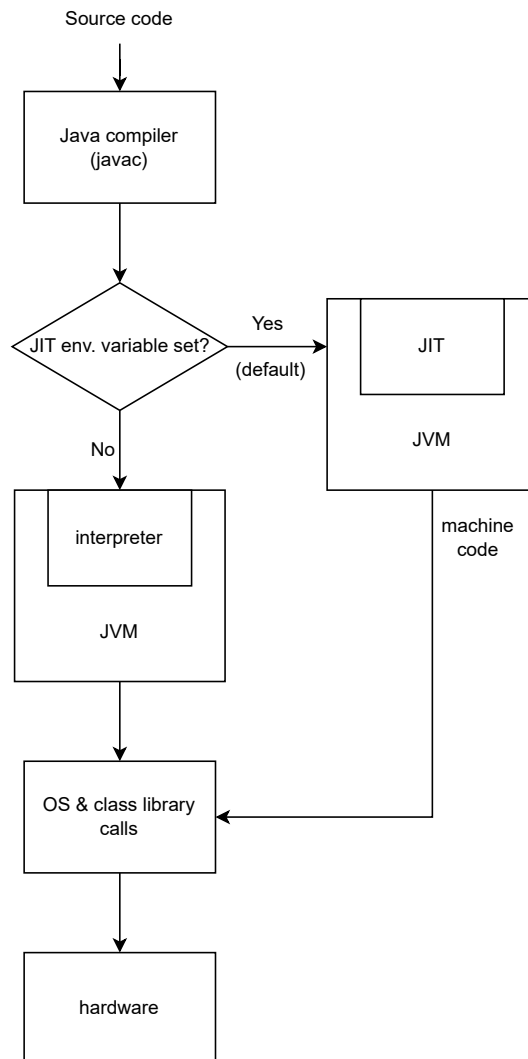


Figure 2.1: A diagram of the Java compilation and interpretation/JIT compilation. [28]

2.1.3 OpenJDK and HotSpot

OpenJDK (JDK standing for Java Development Kit) is the official open-source implementation of Java [20]. It is licensed under the GNU General Public License v2.0 [6] and has many contributors from diverse backgrounds/companies.

HotSpot is the default open-source implementation of the JVM and is developed by the OpenJDK group. The HotSpot codebase has been under development for the past 20 years and consists of around 3,000 C/C++ header and source files totaling more than

1.2 million lines of code. It contains a class loader, bytecode interpreter, JIT compilers, and also garbage collectors. [19]

2.2 Garbage Collector

In general, a garbage collector (GC for short) is a part of a programming language’s runtime which is used for automatic reclamation of dynamically allocated objects when they are no longer required during a program’s run. They have been incorporated into many modern programming languages, including Java [30]. The process of allocating and deallocating space for objects is, therefore, automatic with them, and the user of a programming language with garbage collectors can use the language much more easily, without the need for using manual space allocation like `malloc()` in C.

Dynamic object allocation has many benefits because it allows us to store objects in a heap, not on the stack. Heap allocation offers many benefits. It allows programs to dynamically choose object sizes and use recursive data structures like lists or trees. It also makes returning methods from other methods possible (like closures, streams, etc.). Garbage collectors usually manage this by using a pre-allocated chunk of memory throughout the program’s lifecycle. Since garbage collectors manage the memory dynamically, they only reclaim objects when there are no pointers (references) pointing to them from other outside objects, making sure that no objects are deleted prematurely. [11]

Also, in general, garbage collectors are regarded as a safer choice than manual allocation, since there are more memory protection mechanisms in place [15], thereby minimizing potential vulnerabilities that might be more prevalent in languages with manual memory management.

In HotSpot, specifically, there are multiple garbage collectors available for the user to choose from. The choices include the Serial and Parallel garbage collectors, that are mainly used in smaller applications as they are much simpler than their modern counterparts (Serial has a single GC thread, whereas Parallel has multiple), Concurrent Mark Sweep collector (which is designed for applications that require low pause times and have many live objects), and the Garbage-First garbage collector, which was designed for modern workloads and works great on large heaps with a mix of objects (both short and long-lived) and it is also the de facto default garbage collector in HotSpot (de facto, since there are situations where the JVM might choose a different one if none is specified by the user and the JVM concludes that another GC would fit the given situation better, but those situations are relatively rare). [8]

There are also other newer garbage collectors available in HotSpot (such as Shenandoah [5] and ZGC [22]), but the thesis will focus solely on the Garbage-First one.

2.3 Garbage-First (G1) Garbage Collector

As described on Oracle’s website, “G1 is a generational, incremental, parallel, mostly concurrent, stop-the-world, and evacuating garbage collector which monitors pause-time goals in each of the stop-the-world pauses” [24].

Below is the explanation of the adjectives in this sentence in simpler terms: [24]

- *generational* – The garbage collector splits its memory into two main categories (or generations), young and old. New objects are typically allocated to the young generation, and only when they survive multiple GC cycles are they “promoted” into the

old generation. This is done because objects die young most of the time (in the first few GC cycles), so it makes sense to scan the young generations more often. [4]

- *incremental* – The garbage collector can do its tasks in incremental pieces. This is helpful because doing the garbage collection as a single big operation can significantly slow down the program execution.
- *parallel* – The garbage collector can use multiple threads to perform its tasks.
- *mostly concurrent* – The G1 garbage collector does *most* of its tasks concurrently with the running program, without stopping it. This also speeds the program, because it will only be stopped when really necessary (in the “stop-the-world” phase).
- *stop-the-world* – An event when the GC freezes the execution of the program to do the cleaning tasks, which cannot be run concurrently with the program. The goal of the GC is to minimize these stop-the-world pauses and only stop for a minimal time.
- *evacuating* – Evacuation (or “copying”) stands for a task when the garbage collector evacuates (or copies) live objects to different regions in the memory (which are currently unoccupied) and then reclaims the old region for future use. This part of the cycle is potentially dangerous. Objects are copied rather than literally “moved”. As a result, multiple copies of the same object might exist in memory, and they could contain sensitive information. [4]
- *monitors pause-time goals* – The garbage collector has the ability to monitor the length of the stop-the-world pauses and allows users to set their desired maximal pause times, which the GC will then try to adhere to.

2.3.1 Memory Layout

As already mentioned above, the G1 GC has its pre-allocated memory, which it splits into equally-sized chunks called “heap regions”. The size of the memory or the number of regions depends on a few settings and can differ in different environments/programs. Generally, the size of one heap region ranges from 1 MB to 32 MB, and there can be up to 2048 of them [1].

At any point in a program’s execution, each of these regions can either be empty (unassigned) or assigned to the young generation or the old generation. As requests for object allocation (or move) come, the G1 GC hands out the free memory regions, assigning them to one of the generations, and they serve as a free space for the allocation (or object movement). [24]

The young generation regions are further split into two categories, the *eden* regions and the *survivor* regions. When a new object is created, it is automatically allocated in one of the eden regions (it is most likely to be collected in the next cycle). If the object survives a young generation collection (this collection is only focused on collecting garbage in young regions, it is sometimes called a minor collection), it is moved into one of the survivor regions. The survivor regions make up two survivor spaces, the S0 and S1. [10]

When a minor collection is triggered, it looks for garbage in eden regions and in the current active survivor space. The objects that survive this collection are copied either to the other (currently inactive) survivor space or, if they were already in a survivor region and survived enough collections (this number can change depending on the environment

and setting), are promoted and copied to an old region. It is assumed that such object is “aged out” and that it will survive for a while there. After the minor collection, the roles of the S0 and S1 survivor spaces are swapped. The one that was active becomes inactive, and the inactive one will be active. [26]

There can also be special regions called *humongous*. These regions belong to the old generation, and they are meant for really large objects, which typically span multiple heap regions (but the minimum for an object to be humongous is having the size of half of a region). The G1 treats those objects a bit differently than the rest, they get allocated in the old generation by taking multiple continuous regions and putting the object right at the start of the first of those regions. The leftover space from the last (non-full) region is being unused until the object is deallocated. The allocation of humongous objects might trigger the GC prematurely, since they can trigger the heap occupancy threshold, triggering the GC very easily. [24]

Figure 2.2 shows an example layout of the memory when using the G1 garbage collector. The grey blocks with the “U” letter represent empty heap regions (without any assigned object or even a generation). The blocks with an “E” letter represent eden regions (the place where new objects will be allocated). The blocks with a dark yellow background with the letter “S” represent survivor regions. Finally, the blocks with a green background and the letter “O” represent old regions. Humongous regions are ignored in this graphic.

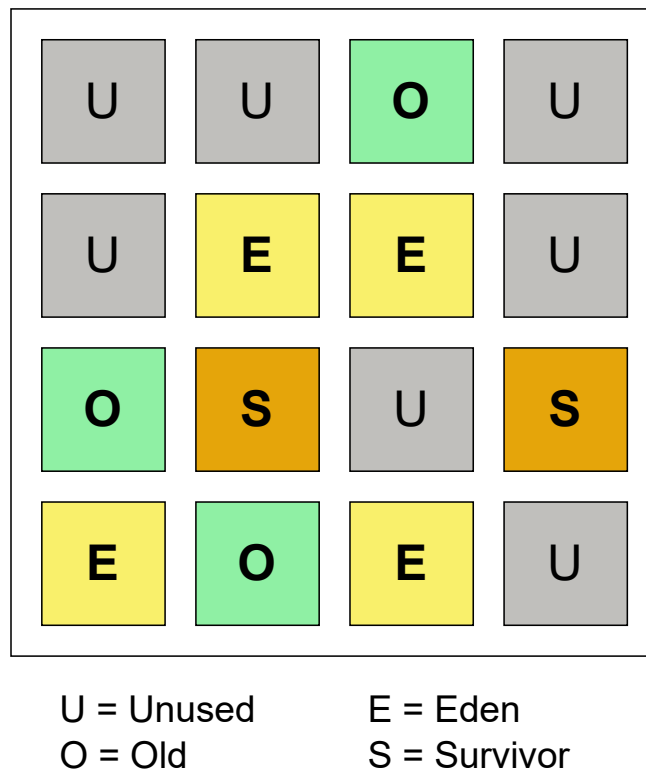


Figure 2.2: An example of the G1 GC heap layout.

2.3.2 The G1 GC Collection Cycle

On the higher level, the G1 garbage collector has two main collection phases between which it alternates – the *Young-only* phase and *Space Reclamation* phase. In the Young-only phase, only objects from the young regions are collected and are gradually promoted into old regions when surviving multiple of these collections. On the other hand, in the Space Reclamation phase, even objects in the old regions are being collected (together with objects from young regions). After the Space Reclamation ends, the garbage collector continues with the Young-only phase. [24]

This whole collection cycle is shown in Figure 2.3.

Young-only Phase

This phase begins with several normal young collections, during which some objects are promoted to the old generation. The transition from the Young-only phase to the Space Reclamation phase starts when the size of objects in old regions exceeds a certain threshold (known as the Initiating Heap Occupancy Threshold). At that point, instead of performing a normal young collection, the G1 garbage collector schedules a Concurrent Start young collection. [24]

The Concurrent Start collection begins with a *marking process* together with a normal young collection. This concurrent marking phase identifies all reachable (live) objects in the old generation regions, making sure this information is ready for the upcoming Space Reclamation phase. During this marking, normal young collections can still occur. The marking process ends with two stop-the-world pauses, Remark and Cleanup. [24]

The Remark pause “finalizes the marking itself, performs global reference processing and class unloading, reclaims completely empty regions and cleans up internal data structures” [24]. After Remark and before Cleanup, G1 calculates which old generation regions can be reclaimed concurrently. The Cleanup pause then chooses if the Space Reclamation phase will proceed. If it does, the Young-only phase ends with a Prepare Mixed young collection. [24]

During the Young-only phase, the G1 GC continues to maintain region granularity, isolating young region activity from the old regions unless necessary. This is done to keep pause times low by avoiding potentially expensive whole heap scans and only promoting objects that truly need to survive. The promoted objects are likely to be living for a while, so it is unnecessary to scan them so often.

Space Reclamation Phase

The Space Reclamation Phase starts after the Young-only phase. In the Space Reclamation phase, multiple mixed collections occur, which, in addition to objects in young regions, also collect objects in old regions. Also, in this phase, both live objects in the young and old regions are being evacuated between regions, yielding space. This phase ends when the garbage collector determines that evacuating more old regions would not be worth the effort (not enough space would be reclaimed), and G1 then continues with the Young-only phase [24].

This phase aims to reclaim as much space as possible, so the program can continue without a bigger interruption for as long as possible. By cleaning the old regions, the G1 GC makes space for new objects that will be promoted to one of them. Also, there is a backup mechanism if the G1 GC runs out of memory during the process of checking the

objects to be collected, it will perform an in-place stop-the-world full heap compaction like other GCs do, leading to freeing memory at the price of worse performance. [24]

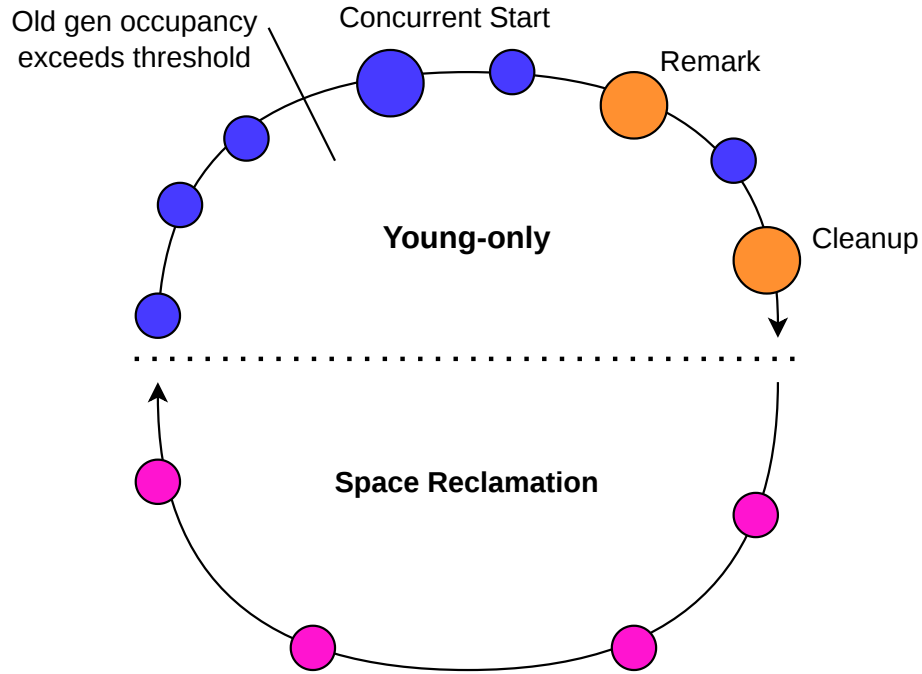


Figure 2.3: A figure showing the G1 GC cycle, based on an image from [24].

2.3.3 Remembered and Collection Sets

When a garbage collector does not scan the whole heap at once (as in the Young-only phase), it must track references that cross the boundary between the part of the heap that is not being collected and the part that is. This commonly occurs in most generational garbage collectors, where the uncollected region is usually the old generation and the collected region is the young generation, which is also the case here. The structure used to store this information, meaning references from old generation objects to young generation objects, is called a *remembered set*. [25]

To ensure that the G1 is as efficient as possible, these remembered set entries are represented as references to so-called *cards*, which are logical partitions of the heap, typically 512 bytes in size. This allows a single entry (card) to represent multiple objects, reducing the memory usage. G1 GC manages these remembered sets by grouping regions that are beneficial to be collected in the same collection, and during the garbage collection process, it generates a single big remembered set for all regions selected for evacuation. This process involves removing duplicate entries and deleting unnecessary data to maximize evacuation efficiency. While the remembered sets for the young generation regions are maintained continuously (due to their frequent collections), those for marking collection set candidate regions are rebuilt between the Remark and Cleanup pauses. [24]

On the other hand, the *collection set* identifies the specific source regions targeted for space reclamation during a garbage collection cycle. Regardless of the collection type, this set includes young generation regions, humongous regions, and specific old generation regions selected as collection set candidates. G1 identifies these candidates based on their collection efficiency, which is determined by the amount of free space available and the degree of connectivity (number of references) to other regions. High efficiency is typically found in regions with minimal live objects and low connectivity. [24]

These candidates originate either from whole heap analysis, which provides updated object liveness data, or from regions that previously had an evacuation failure (explained in section 2.3.5). Because regions with evacuation failures often contain very few objects, they are highly efficient to reclaim and are included as candidates by default. G1 also distinguishes between mandatory collection set candidates that must be processed in the current cycle and optional candidates that are only collected if the time allows it. [24]

2.3.4 The Garbage Collection Process

On the lower level, the actual garbage collection process in G1 GC consists of four different phases: [24]

- *Pre Evacuate Collection Set phase* – In this phase, the GC does work to prepare for the collection, such as disconnecting Thread Local Allocation Buffers (dedicated blocks of memory within the Java heap assigned to individual threads, allowing them to allocate new objects quickly without the performance bottlenecks of multi-threading [34]) from the mutator threads and selecting the collection set.
- *Merge Heap Roots phase* – G1 combines the individual remembered sets from multiple regions into a single, unified list. By creating this one list of where objects are located, the JVM can remove duplicate information, leading to much faster collection.
- *Evacuate Collection Set phase* – This is where the actual movement of objects in the heap occurs. The collector identifies “root” references, which are pointers (references) from outside the targeted area that lead to objects inside it. G1 then copies these referenced objects to a new location in the heap (often in the old generation regions). It continues this process by following every connection until all reachable objects are moved and no more roots remain to be processed.
- *Post Evacuate Collection Set phase* – The cleanup phase, work like reference processing (identifying and handling special references to objects) is being done in it.

2.3.5 Evacuation Failure

Evacuation failure occurs when G1 lacks space to move objects during a garbage collection cycle, temporarily making the affected regions unavailable for new allocations. To resolve this, G1 prioritizes these regions as collection set candidates for evacuation in the next cycles. In extreme cases where G1 fails to reclaim any space, the JVM triggers a full garbage collection. This fallback mechanism performs an in-place compaction of the entire heap, which is very effective at recovering memory, but often results in significant overall performance loss. [24]

2.3.6 Marking

Marking is the process of checking objects in a collection set and determining whether they are live or not (should be collected or not). G1 uses an algorithm named *Snapshot-At-The-Beginning* (SATB). This algorithm works by taking a virtual snapshot of the heap at the time the Concurrent Start pause occurs and using it during marking. This means that there can be objects that will become unreachable during the marking, but will still be considered live for the space reclamation. This, unfortunately, leads to incorrectly retaining objects that should have been collected, however, it comes with increased performance during the Remark pause. The uncollected, unreachable objects will be collected in the next marking phase. [24]

2.3.7 Initial Heap Occupancy

The *Initiating Heap Occupancy Percent* (IHOP) represents the threshold for triggering a Concurrent Start collection. It is calculated as a percentage of the old generation size. By default, G1 uses Adaptive IHOP to dynamically determine an optimal percentage threshold. This automatic mechanism analyzes the duration of marking and the amount of memory allocated by the old generation during GC cycles to ensure that memory is used efficiently. However, as with almost anything in G1 (it has a lot of configurable options), this threshold can be set by the user to any static value. [24]

2.3.8 Periodic Garbage Collections

To prevent unnecessary memory use during an application's inactivity, G1 can be configured to perform periodic garbage collections. If configured, the JVM monitors the time elapsed since the previous collection and initiates a GC cycle if the application remains idle for the specified time. The type of this collection depends on the current heap phase, it may trigger concurrent marking, a full GC, or continue space reclamation as required. Also, G1 has an option to make sure these operations only occur when the JVM's load is low enough. When the JVM's load is above a certain value, the periodic GC cycle will not be run. [24]

Chapter 3

Potential Memory Vulnerabilities in Garbage Collectors

Currently, the most common vulnerabilities in any software are related to memory mismanagement/gaining unauthorized access to a program's memory [3, 23]. Since the sole purpose of garbage collectors is to manage memory for objects (allocate and deallocate them dynamically), they are also very prone to those memory vulnerabilities, and preventative measures need to be taken to minimize them. The implementation of modern garbage collectors is often also very complex, so there might be a situation where a developer creates or overlooks a human mistake, which could lead to potentially exposing the code to these vulnerabilities. In the worst scenario, an attacker might misuse those bugs to gain access to the JVM's memory, gaining access to various, potentially sensitive data.

In this chapter, the most common memory-related vulnerabilities and concepts are described, namely *dangling pointers*, *use-after-free* vulnerability, *double-free* vulnerability, and *buffer overflow* vulnerability.

3.1 Dangling Pointers

Put simply, dangling pointers are created when the data in memory that a pointer points to (these might be numbers, strings, or even more complex objects) is deallocated, but the pointer still keeps pointing to that (now dead) memory. A pointer to dead memory causes serious issues if it remains accessible. It creates an opportunity for use-after-free and double-free vulnerabilities. There are two main events that need to happen, potentially far away in time, in order for a dangling pointer to be misused in one of those vulnerabilities: the creation of the dangling pointer (an object it pointed to was deallocated or moved elsewhere) and the use of the dangling pointer (access to the new data it now points to or freeing the memory again). [2]

Dangling pointers are active until they are destroyed by setting the address they point to to NULL or by deallocating the memory where they are stored (but new dangling pointers might be created by that) [2].

Below is a simple example of how a dangling pointer can be created and destroyed in a C program, as it is the easiest to understand this issue in a lower-level language. Between the `free(buffer)` and `buffer = NULL`, the `buffer` pointer is dangling, meaning that it points to the memory where the *Hello!* text was, but now, there can be anything, since the memory was freed and reused for other purposes.

```

#include <stdlib.h>
#include <string.h>

int main() {
    // Creating a new pointer to allocated memory.
    char *buffer = malloc(32);

    // Putting some data into the memory.
    strcpy(buffer, "Hello!");

    // Freeing the allocated memory,
    // "buffer" becomes a dangling pointer.
    free(buffer);

    // THE DANGEROUS ZONE
    // If the "buffer" pointer was to be used here,
    // it would potentially create problems.

    // The dangling pointer "buffer" is now destroyed.
    buffer = NULL;

    return 0;
}

```

In the Figure 3.1 below, the lifecycle of the *buffer* dangling pointer is shown. In the first column, it correctly points to the text copied to the memory. The second column shows the situation after the *buffer* was freed, the pointer still exists and points to unknown data, here marked with a question mark. The last column is the state before the **return** statement, *buffer* now points to `NULL`, unable to access any data in memory.

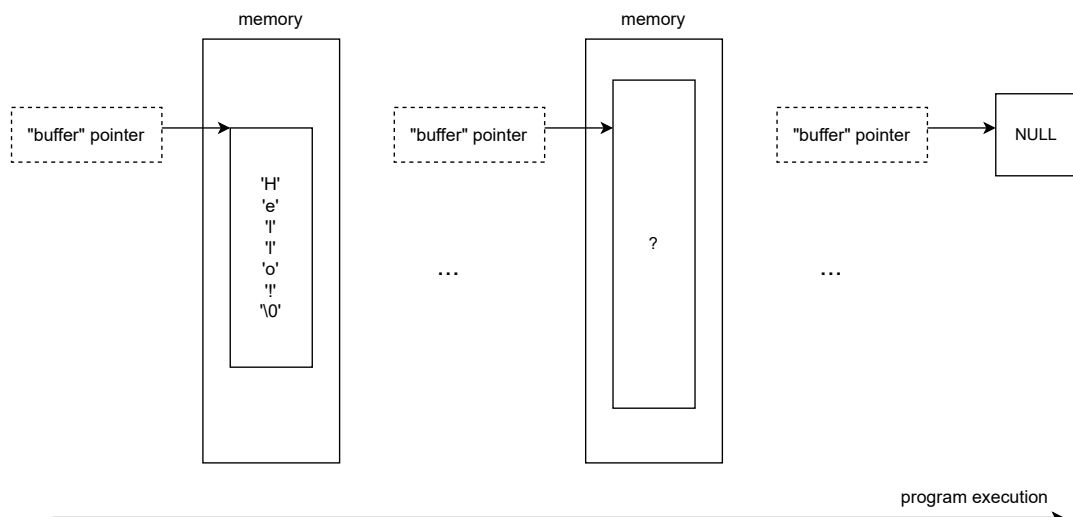


Figure 3.1: A diagram showing the lifecycle of the *buffer* dangling pointer in the program above.

3.2 Use-After-Free Vulnerability

Use-after-free (commonly abbreviated as UAF) is a type of vulnerability that occurs when dangling pointers are misused to access and operate on memory [2]. When a dangling pointer is used to access memory, the data it points to may be exposed and even manipulated. Since the data may now belong to a different pointer, modifying it with the first one may cause unexpected behavior or crashes. An even more severe situation may happen when the newly allocated data in that place is a class (in C++, for example). If the class contains a pointer to a function and that pointer is changed to point to a valid, arbitrary shellcode (potentially set up by an attacker), it may be executed. [17]

In the Figure 3.2, the typical simplified process of this vulnerability is shown. It starts with *pointer 1* pointing to an *object 1*. Then, the object is deallocated and *pointer 1* becomes a dangling pointer. Then, the space where *pointer 1* is pointing becomes occupied by an *object 2* referenced by *pointer 2* (the space was freed, so it is okay for other objects to be allocated there). Since the first pointer still exists, it can access and modify data inside the second object, leading to an exploitable situation if the first pointer is somehow accessible from the outside, for example, by some interface.

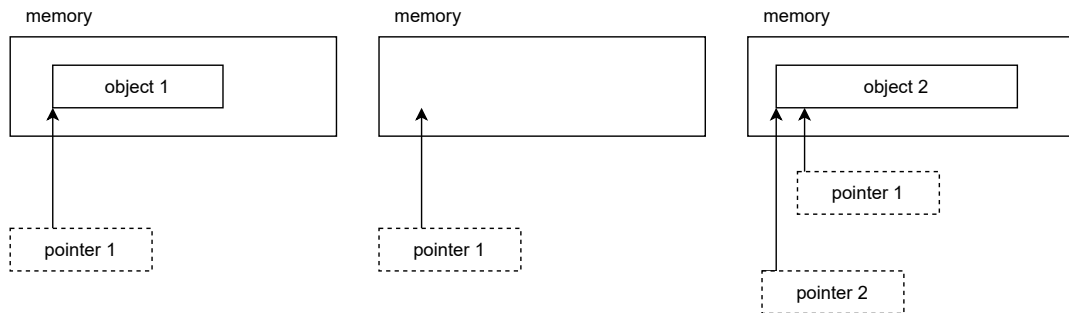


Figure 3.2: A diagram demonstrating the use-after-free vulnerability.

Let's take a look at an example of this vulnerability on an edited example from the section above:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void printHex(char *pointer) {
    // Prints the 32 bytes starting from a given pointer.
    // The data is printed as hexadecimal numbers.
    for (char i = 0; i < 32; i++) {
        printf("%02x ", (unsigned char) pointer[i]);
    }
    printf("\n");
}

int main() {
    char *buffer = malloc(32);
    strcpy(buffer, "Hello!");
```

```

    // Print the buffer with "Hello!" text inside.
    printHex(buffer);

    free(buffer);

    // Use-after-free: Print the values that are
    // now in the memory where "buffer" points.
    printHex(buffer);
    // Copy the "Hello!" back there.
    strcpy(buffer, "Hello!");
    // And print the "buffer" again.
    printHex(buffer);

    buffer = NULL;
    return 0;
}

```

The output of this program might look like this (the values on the second and partly the third lines will be random):

```

48 65 6c 6c 6f 21 00 00 00 00 00 00 00 00 00 00 ...
93 52 02 00 00 00 00 00 2c 32 87 82 00 85 a7 34 00 ...
48 65 6c 6c 6f 21 00 00 2c 32 87 82 00 85 a7 34 00 ...

```

As can be seen, the first line contains only hexadecimal values for the string *Hello!*, followed by zeros, which is the expected behavior, the *buffer* pointer is still valid at this point. The second line contains seemingly random data because the memory there was freed and is now being reused for other purposes. While this might seem okay, in more complex examples, this data might belong to another pointer, and it might even be something the first pointer was not supposed to see. Between the second and third lines, the *Hello!* string was put back in the freed memory, overwriting whatever was there before. Again, the overwritten data might belong to different pointers, leading to undefined behavior, crashes, or, in some very specific cases, to code execution [17].

This low-level example should make it clear that this vulnerability is highly dangerous when present. However, it might be even more dangerous when present in garbage collectors for higher-level languages, as the language's users might not even be aware that this vulnerability exists there. Let's imagine there is a vulnerability in the G1 GC that leads to creating a dangling pointer to an object that was deallocated or moved. Since G1 GC constantly moves objects to different places in memory, let's imagine that an object containing user passwords ended up in the place of the old object. Now, if the user (or an attacker) has a way to access this dangling pointer (for example, through a reference to that object), they might be able to read those passwords or even change them, leading to a severe leak of sensitive information. Also, it is worth noting that when objects are deallocated and moved, the physical data itself remains in its old location until overwritten by another object (it would be very time-consuming to clear the physical bytes when an object is moved or deleted). So the sensitive information can potentially stay there for a long time, leaving the window for exploitation open.

3.3 Double-Free Vulnerability

The second vulnerability type that might occur when dangling pointers are present is a so-called *double-free vulnerability* [2]. In traditional low-level languages, double-free occurs when `free()` is called twice on the same pointer to memory. This might corrupt a program's memory management structures, which in turn can lead to reading or modifying unexpected memory addresses. This might again lead to a crash of the program or, in worse cases, for `malloc()` to return the same address twice, opening the door for an attacker to write data into a data structure they should not have access to. If this structure were a C++ class again, they might be able to modify the function pointers to point to arbitrary code for execution. [16]

An example of how this vulnerability might occur is shown below:

```
#include <stdlib.h>
#include <string.h>

int main() {
    char *buffer = malloc(32);
    strcpy(buffer, "Hello!");

    // Creating a copy of the pointer.
    char *bufferCopy = buffer;

    // Trying to free the memory two times.
    free(buffer);
    free(bufferCopy);

    buffer = NULL;
    return 0;
}
```

On modern Linux systems, this simple example would not work, as glibc has a mechanism in place that catches the attempt to free already freed memory, leading to a core dump. However, if we were to combine this example with the use-after-free vulnerability and add a `strcpy(buffer, "aaaaaaaaaaaaaaaaaaaaa")` call between the `free()` statements (filling the buffer with custom data), the program executes "correctly", it does the call to free the memory twice, leading to unexpected behavior.

Since this vulnerability is very low-level and tied to manual memory management, it is unlikely to occur in garbage collectors on its own, as they most of the time allocate a large chunk of memory to use throughout the program's run. Allocating and deallocating physical memory is mostly done at the start and end of the program, respectively. However, similar unexpected behavior can happen even in them. Let's say that some garbage collector deallocated an object, but a dangling pointer to it still exists and is exposed. If a handy attacker gets a way to deallocate this (now already gone) object again, it might also lead to very unexpected behavior, such as a deallocation of a live object. While this is a very hypothetical situation, preventative measures would look very similar to those for the use-after-free vulnerability, so it is worth mentioning and knowing about.

3.4 Buffer Overflow Vulnerability

According to OWASP (Open Worldwide Application Security Project), buffer overflow occurs “when a program attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer”. By buffer, a continuous chunk of memory containing multiple individual pieces of data is understood (for example, an array of integers or a string). [29]

When a program tries to put data beyond the given buffer, many different situations may occur, since the data there is unknown and may belong to other structures or buffers. Of the many situations that may happen, the most common ones are data corruption, crashing the program, or executing malicious code. It is probably the best-known type of security vulnerability in software, and it is also considered very high severity with a very high likelihood of exploitation when it is discovered in a given piece of software. [29]

A simple example is, again, shown below. It extends the use-after-free example and contains two notable main changes. The first one is that the `printHex()` function reads 40 bytes, meaning that it also accesses bytes after the allocated buffer and reads them (which in itself is an access out of the permitted bounds). The second main change is the line containing `buffer[35] = 'a'`. This essentially puts a character into memory 35 bytes after the place where `buffer` is pointing, which is a problem, because the allocated space for `buffer` is only 32 bytes long, meaning that buffer overflow happens. In more complex scenarios, this might lead to overwriting data belonging to a different pointer.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void printHex(char *pointer) {
    // Prints the 40 bytes (8 bytes more than buffer holds)
    // starting from a given pointer. The data is printed
    // as hexadecimal numbers.
    for (char i = 0; i < 40; i++) {
        printf("%02x ", (unsigned char) pointer[i]);
    }
    printf("\n");
}

int main() {
    char *buffer = malloc(32);
    strcpy(buffer, "Hello!");
    // Simple buffer overflow, putting a character after
    // the allocated 32 bytes.
    buffer[35] = 'a';
    // Print the buffer and data after it.
    printHex(buffer);

    free(buffer);
    buffer = NULL;
    return 0;
}
```

The output (split into two lines) of this simple program may look like this:

```
48 65 6c 6c 6f 21 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 61 00 00 00 00
```

The first few bytes are the hex values of *Hello!*, but the 61 (belonging to the 'a' character) is clearly shown there, on the 36th position (arrays index from 0). This is also demonstrated in Figure 3.3. The `buffer[0]` pointer (equivalent of `buffer`) points to the start of the allocated 32 bytes of the buffer (the whole rectangle is considered the *buffer*), whereas the `buffer[35]` pointer points outside of the buffer, meaning that it overflowed.

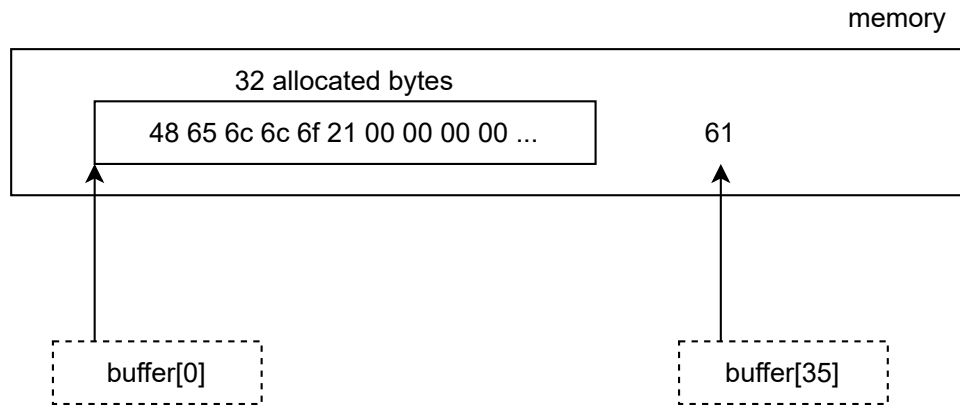


Figure 3.3: A diagram demonstrating the buffer overflow vulnerability.

As with the other vulnerabilities described here, it is primarily a vulnerability that appears in lower-level languages such as C/C++ or Fortran. There are measures in place for catching accesses outside of bounds for a given array in Java, Python and other high-level languages, making these languages immune to this type of vulnerability (for example, in Java, an `IndexOutOfBoundsException` exception will be triggered, when trying to access an index which is beyond the size of a given array or string). However, the implementations of these languages are often done in lower-level languages, so this vulnerability (and the others mentioned above) might still occur in their interpreters, or, the main topic of this thesis, their garbage collectors.

If there is a bug present in a garbage collector, allowing a potential attacker to gain access to an overflowable buffer, they might be able to gain access to information they should not have access to (or even modify it). Most garbage collectors, including G1 GC, treat memory as one continuous chunk. If a buffer overflows, the entire heap becomes exposed because it lacks any divisions.

Chapter 4

Planning SanitizeGC Improvements

This chapter focuses on planning the improvements that can be made to the Garbage-First garbage collector to minimize the threat of vulnerabilities described in the previous chapter. The goal is to design changes to the underlying garbage collector logic (focusing on the memory management) that would help with catching dangling pointer accessing memory that is not theirs, and also catching buffer overflows. The idea is to take one platform (in the case of this thesis, JDK 23 on Linux x86-64) and design improvements that are implemented in the next chapter, serving as a proof-of-concept for further research and potentially opening a discussion about adding those changes to the HotSpot code.

4.1 Inspiration

This project takes direct inspiration from Google’s AddressSanitizer (or ASan). This sanitizer is a memory error detection tool for C/C++ languages, and it works by replacing the `malloc()` and `free()` with their custom implementation. Their `malloc()` not only allocates a given chunk of memory, but it also allocates so-called *red zones* around it. These red zones are inaccessible, and they serve to catch buffer overflows or underflows, so when there is a pointer trying to access them, the sanitizer will catch that. Also, when using `free()` on the chunk, the sanitizer will also make the previously accessible chunk inaccessible, meaning that any access to it will also be invalid (effectively minimizing dangling pointer vulnerabilities). The `malloc()` and `free()` functions also have a mechanism in place to track the current call stack, so that debugging is easier. [33]

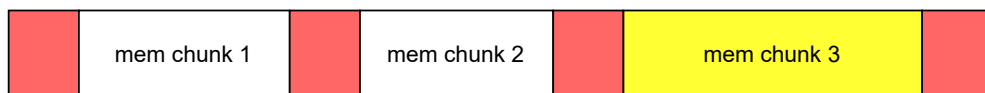


Figure 4.1: A diagram demonstrating AddressSanitizer’s memory layout.

Figure 4.1 demonstrates its functionality on an example memory layout. The red rectangles (without any text) represent the inaccessible red zones, any access to them will result in an error. The white rectangles represent currently allocated and in-use memory chunks, they are normally accessible and okay for use. The yellow rectangle (“mem chunk 3”) is a memory chunk previously used, but it was freed, and now it is also inaccessible. These

two mechanisms combined serve the purpose of catching buffer overflows and use-after-free errors, effectively minimizing them at the cost of a 73 % slowdown and a 3.4x increased usage of memory [33].

4.2 The Main Idea

The main idea to improve the security in the G1 garbage collector is to move the memory regions it works with to new places in memory multiple times throughout the program's lifecycle. As was described in Chapter 2, G1 has a preallocated chunk of memory it uses as the heap, and it splits it into multiple *heap regions*. All of these heap regions start as free regions (meaning that they do not contain any data and have no generation) in a so-called *free region list*, and throughout the run of a program, G1 takes them from the list when it needs them for object allocation. Also, when a garbage collection happens, all of the now-empty regions are cleared and returned back into the free region list.

What we want to do is locate the exact place where the heap regions are taken from the free region list and call a special method that will allocate a whole new memory for it (by using memory mapping methods like `mmap()`, which creates a new memory mapping within the virtual address space of the process that called it [13]), edit the heap region structure to point to that new memory, and lock the old place, where the region was (by using methods like `mprotect()`, which modifies the memory protection settings for any pages within a specified address range [14]). Accessing the old memory triggers a page fault. The JVM catches this fault and provides a stack trace showing the exact location of the error. That means that if there was some dangling pointer actively accessing the memory, the program will crash, leaving logs for inspection. Also, since the garbage collection happens relatively often, the addresses of the heap regions (especially containing the young generation objects, which are more subject to creating these vulnerabilities, since they are allocated, moved, and deallocated often) will change every time they are freed and used again. This will create a continuous loop of region movement, potentially catching newly created dangling pointers.

In Figure 4.2, the moving and locking process is shown. Each column represents the memory, the one on the left is at the start of a JVM, the middle one is after some regions were used/moved, and the one on the right is when more regions were moved, some even twice. The rectangles with blue filling and no text represent heap regions (only 6 in this example for easier understanding), and the red rectangles with “x” represent locked memory. As can be seen, at the start, the JVM allocates all regions into a single big chunk next to each other. However, with our extensions, this memory will never be touched. When G1 tries to take one of those regions to allocate objects into it, they will be moved into a different place (in this example, not all regions are immediately used, so some stay in the same place, indicated by a solid arrow), leaving the original memory they were located on locked.

Once a region has been moved at least once, it will stay in the place until G1 evacuates all objects from it somewhere else, returning it to the free region list. At this point, the old memory left behind will also be locked, and once the region is reused, it will be moved to a new location.

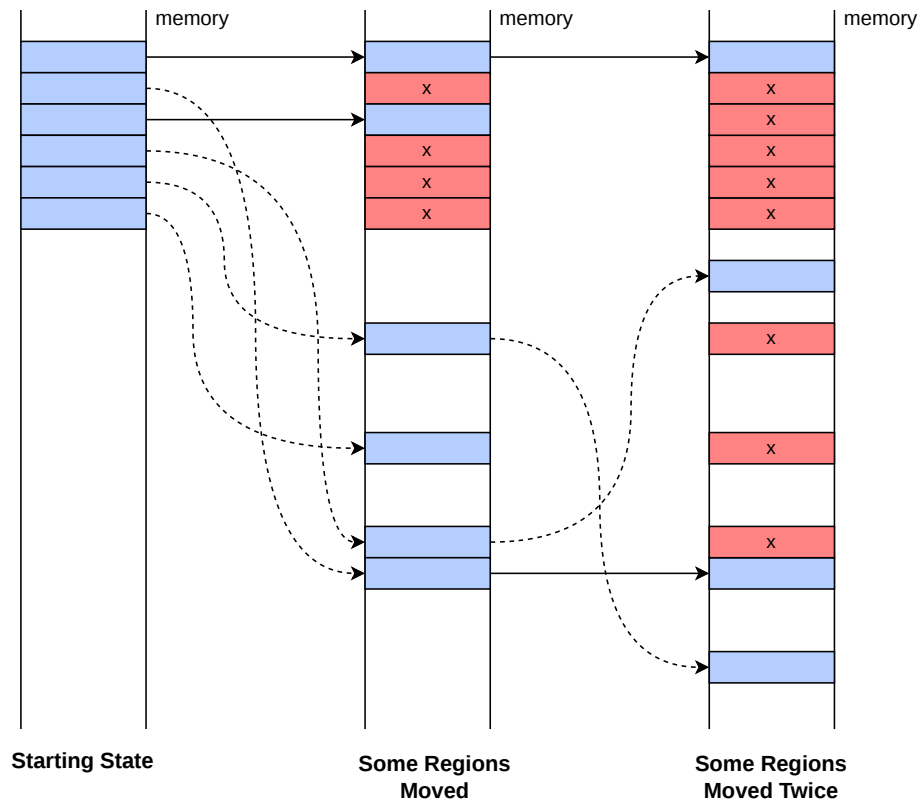


Figure 4.2: A diagram demonstrating how the region moving might work in a heap.

4.2.1 Card Table Problem

A card table is a specific kind of structure that stores cards that represent the heap memory blocks with pending refinement (meaning they contain modified object references that are waiting to be scanned and processed). The Java HotSpot VM implements it as an array of bytes, where each byte is known as a card. Each card represents a range of addresses in the heap. “Dirtying” a card means updating its byte to a special value that signals there may be a pointer from the old generation to the young generation within the address range covered by that card. [25]

The problem with this approach to these card tables is that the addresses the card tables correspond to are calculated using a base heap address and bit shifting based on offset from that base address. But since the regions will move constantly, the offset will change, invalidating those calculations.

And card tables are not the only problematic structure, there are a lot of them. Luckily, most of them have one thing in common – they are a bitmap-like objects where each unit in them (bit, byte, etc.) represent a heap block. This approach assumes that the heap is continuous and the regions are laid out together. However, with our changes, that will not be the case, the regions will be spread out throughout the memory.

This is why there needs to be a structure created that would help with the translation from the original heap address (used for calculations where necessary) to the new heap address (used for actual memory access) and vice versa.

4.2.2 Address Mapper

The structure for solving this problem (simply called mapper from now on) will work by taking an address that needs remapping, masking away its lower bits (but saving them for later use), leaving only the starting address of a region to which the address points. Next, it will get the corresponding remapped starting address of the region and add those saved lower bits again to get the whole remapped address pointing to a real place in the region.

The G1's heap regions are aligned in memory and there is a global variable representing the number of lower bits in each address that correspond to the addresses inside of a single region, so we can take this number to know how many bits of each address we should mask (the regions might have different sizes, so this number cannot be hard-coded) to leave only the starting address of a region.

For the most efficient address remapping, a *hash map* was chosen to be used as the structure that will hold the starting addresses of the regions and their corresponding remapped values. Since in most cases, there are not too many heap regions (around 2000 most of the time), using hash maps for storing them should be very quick with minimal hash collisions.

Since this operation needs to be doable in both ways, there will be two hash maps, one for mapping from original heap addresses (those that are allocated when the JVM starts) to new “moved” addresses (the addresses to which the regions will be moved as part of the extension) and one for the other way.

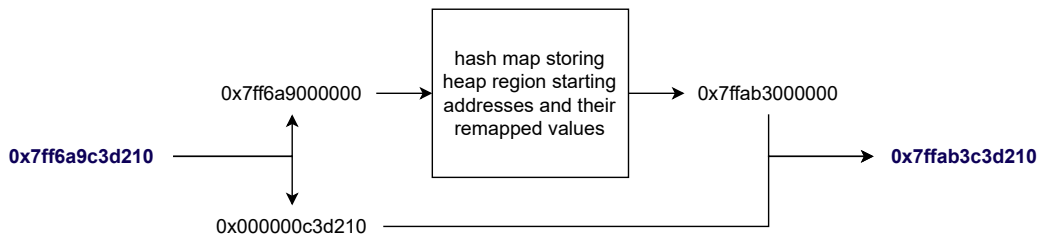


Figure 4.3: A diagram demonstrating the mapper's functionality.

The process is explained with an example in Figure 4.3. The input address to remap is `0x7ff6a9c3d210`. Let's say this address is within a newly allocated region, points to an object, and needs to be remapped to its original address in the original heap address space the JVM allocated at the start, so it can be marked as dirty in the card table. Also, a global variable is set telling us that the lower 24 bits (6 characters in hex addresses) of every address belong to the space of one heap region. Because heap regions are specifically aligned in memory, this means that we can get any region's first address by taking those lower 24 bits away.

So in this example, when we mask away those lower 24 bits, we are left with address `0x7ff6a9000000`, which is the starting address of the region this address is in. The masked lower bits (`0x000000c3d210`) represent the offset from the start of the region to the object inside of it. Since we update the hash map with every region move, we can find the corresponding remapped starting address within it when inputting `0x7ff6a9000000`. In this case, the corresponding “original” address for this region is `0x7ffab3000000`. The

only thing left is adding those offset lower bytes (since region sizes are always the same, the offset to the object in the moved region would be the same as in the original region), leaving `0x7ffab3c3d210` as the resulting remapped address that can be used for manipulation with the card table.

4.2.3 Locking of The Old Space

As already explained, the old memory left behind after regions are moved needs to be locked so that dangling pointers are actually caught. There will be two slightly different mechanisms, one for locking the originally allocated memory for regions allocated by the JVM (in which no actual objects are ever allocated) and one for locking the memory behind regions that were already moved at least once (in these, there actually were objects).

For the originally allocated regions and their memory, when a region is about to be first used, new memory for it will be allocated, and the original old one will be locked in the same place using the `os::protect_memory()` method (inside this method on Linux, it uses standard `mprotect()`). This method is already built into the HotSpot code, and it takes an address, how many bytes to protect, and the type of protection. When locking the original region memory, `os::MEM_PROT_NONE` will be used, indicating that reading and writing to this memory is prohibited, any attempt to do these operations will result in a crash induced by the operating system that the JVM will catch, exiting while writing a comprehensive log with a stack trace.

On the other hand, a slightly different situation will occur when locking already at least once moved regions. Since the time between returning a region to the free region list and taking it again from the free region list is non-deterministic and might be quite long, the actual memory locking needs to be done right at the point when the region is returned to the free region list. There are actually two points where regions are returned to the free region list (they are described in Chapter 5), so both should use the same mechanism.

Problem With Memory Space

While moving regions to new places and locking their old memory sounds great in theory, it is very memory-intensive, since G1 returns and fetches regions from and to the free region list quite often, and the old memory is never unmapped, just protected. To mitigate this, a better way for locking the already-moved-at-least-once regions will be used.

For this, a type of memory managed by the Linux Kernel known as *anonymous memory* will be utilized. Anonymous memory refers to memory mappings that are not backed by any file or device. It is created explicitly by a `mmap()` call, and as an optimization, the kernel allows it to remain anonymous until a user performs a write operation to this memory. Once a write is performed, an actual physical page is allocated for the memory, but until then, it exists only in the virtual address space, without wasting physical space (when a read operation is performed before the first write to that page, it accesses a special physical page containing all zeroes). [35]

What is great about this is that the memory protection rules still apply even when no physical memory is allocated, meaning the memory can be locked (with protection set to `NONE`) without unnecessarily wasting the physical space. Besides the internal kernel bookkeeping overhead required to manage these mappings, the only space that will, by design, be wasted is the virtual address space that every program has (since the goal is to catch dangling pointers, reusing virtual address space would be counterintuitive, as the potential pointer might still exist pointing there). Modern machines operating on

the x86-64 CPU architecture use 64-bit long virtual addresses, but only the lower 48 bits are actually utilized in the current implementations of the architecture, leading to a theoretical maximum address space of 256 TiB (2^{48} bytes) [38], which should be more than enough to run a normal application for quite some time.

In the actual HotSpot code, there exists a method called `os::uncommit_memory()`, which uses this mechanism (it uses a `mmap()` internally with protection set to `PROT_NONE`), so this method will be used in every place where regions are returned to the free region list. However, the original region memory (the one allocated at the start by the JVM) cannot be uncommitted due to some checks in HotSpot, so the memory will only be locked as described above. This means that at any time during the program's execution with these changes, the maximum physical memory used will be twice the size of all regions (twice the size of the same program without the changes).

4.2.4 The benefit of our approach

In Chapter 3, three memory-related vulnerabilities were mentioned: the use-after-free, double-free, and buffer overflow. The changes proposed in this chapter are designed to improve the diagnostic detection of all of those by catching wrong accesses to memory. Both use-after-free and double-free are vulnerabilities caused by a dangling pointer pointing to the memory. By constantly moving heap regions (using the G1's GC cycles and region evacuation) to new places in memory, the dangling pointers will stay pointing to the old place. Because the old place will be locked (stripped of read and write privileges), when the pointer points there, dereferencing it (trying to read or write to the memory) will cause the operating system to catch it. The JVM will get the signal, and it will end its execution, printing the stack trace to a crash log file.

So by doing the region moving, the visibility of use-after-free and double-free vulnerabilities is greatly enhanced. This makes these bugs much easier to debug when and where they occur during development. However, this all relies on the inbuilt GC mechanisms because forcing new collections or evacuations would be very time-intensive and would make the execution significantly slower. G1 evacuates young generation regions more often than old generation regions. Therefore, young regions will move more frequently. This is not a problem because we believe that dangling pointers usually appear in young regions where objects are constantly created and moved.

The region moving may also function as a debugging aid for the third-mentioned vulnerability, the buffer overflow, in some special cases where it is reproducible. Normally, all of the regions are next to each other, meaning that if there exists a way for an attacker to potentially exploit a pointer to overflow (or underflow for that matter), they might be able to scan the whole heap. When SanitizeGC is employed as a dynamic analyzer, the regions get scattered throughout the available memory randomly, and each region will get moved to a new place each time it is evacuated and used again. This intended behavior aims to reveal out-of-bounds errors (when the pointer leaves the region's memory, it will most likely not point to a different region but to unused memory). By intentionally causing a crash and printing a stack trace when a pointer hits unbacked memory, SanitizeGC allows developers to pinpoint the root cause of overflows that might otherwise remain hidden during standard execution.

4.2.5 Known Limitations of the Design

This design works with normal-sized heap regions, but it ignores humongous regions that span multiple of those regions. Humongous regions have their own separate logic for allocating free regions for them, and since they are relatively rare to come across in traditional applications, after a discussion, we made the decision not to move those regions and focus solely on the normal-sized regions for now (our goal is to make the argument that these changes are useful, not to catch every edge case). Unfortunately, this way, the JVM with the extension turned on might have some unpredictable behavior when it encounters humongous objects, even leading to crashes. To mitigate this, the user of the updated JVM might make the region size larger with `-XX:G1HeapRegionSize=<size>` command line flag, which might make some previously humongous objects non-humongous, leading to more expected behavior.

Also, in the last few years, an effort has been made to make Java use less memory, for example, by compressing pointers to objects (internally called compressed OOPs). This is a great way to reduce the memory usage, however, the compression of the pointers is a non-trivial operation, and since we need to work with the pointers to remap them, we chose to ignore those, and the compression will be automatically turned off when the changes are active. When the user tries to turn those on with `-XX:+UseCompressedOops` flag, the JVM will print an error message and exit.

4.3 Alternative Approach Considered

During the development phase, our colleague Thomas Stüfe mentioned an alternative approach to the problem of dangling pointers during a consultation. He argued that, with some considerations, the actual movement of regions is unnecessary (solving some of the issues and difficulties while implementing). His potential alternative approach would work by modifying the mechanism for taking regions out of the free region list and locking the memory of regions that are currently in the free region list.

Normally, the regions are taken from the free region list from the list's tail if they are young generation or from the head if they are old generation (or by a specific index in some cases). With this alternative approach, a *last-in, last-out* mechanism would be implemented, and regions returned to the free list will be there until all other regions that were returned before it are taken out again. This would mean that every region would remain in the free region list as long as possible, making it ideal for locking the region's memory while it is there. Also, the user could easily extend the period during which the region remains locked in the free region list by increasing the JVM's heap size, leading to more heap regions and a larger queue.

In Figure 4.4, the JVM's heap and the free region list are shown. The regions with a green background (2, 3, 5, 8, and 9) are currently being used, so they are not in the free region list. The regions with red or pink backgrounds (1, 4, 6, and 7) are in the free list, meaning that they are currently not being used, and the memory belonging to them is locked and inaccessible. As can be seen, once region 9 is cleaned and returned to the free region list, it will be placed at the start of the queue, spending the longest possible time there. Also, once a new region is needed, it will be region 1, as it is placed at the end of the queue and has no other region before it. The memory belonging to it would get unlocked, and it would work as a normal region in its normal memory.

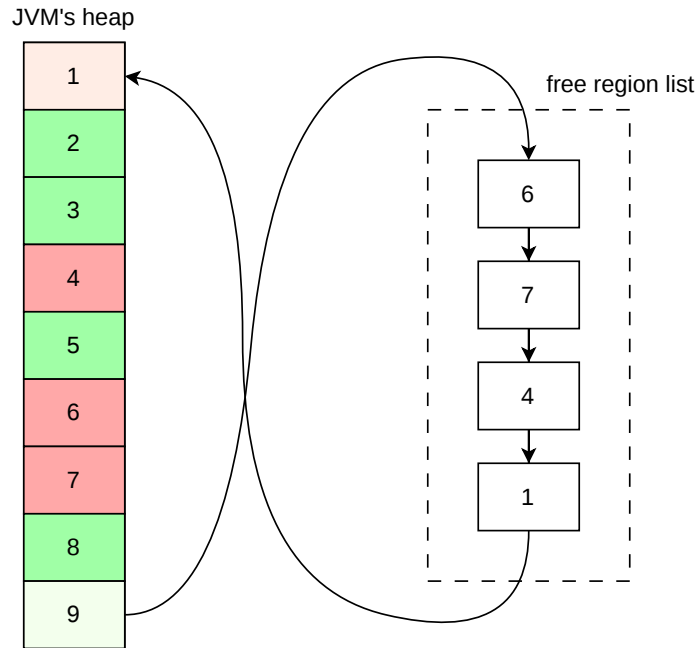


Figure 4.4: A diagram demonstrating the heap and free region list in the alternative approach.

This alternative approach would also help with catching dangling pointers, as the locking mechanism would be the same, meaning that any dereference while the region is locked would get caught. Buffer overflow vulnerabilities would also be at least partly covered because some regions would be locked, so when scanning the heap using buffer overflow, the pointer would eventually reach a locked region, leading to a crash.

While this all seems great, the problem with this approach is that when a region is reused, dangling pointers pointing to it can again become dangerous, as they may point to new objects that will be located in the region. On the other hand, with our first approach, if the regions are reused, they will be located in a completely different part of memory, so the dangling pointers that once existed would no longer be dangerous. I decided to proceed with the first approach for this reason. The alternative design could still inspire future work or serve as a different method to catch dangling pointers.

4.4 Summary of Changes to Implement

In this chapter, the GC improvements were designed with plans for what should be implemented and how. Before moving on to the next chapter, where the actual implementation of those changes is described, let's sum up everything that was mentioned in this chapter, so that there is a list that can serve as a reference later.

- *region movement* – There should be a mechanism that allocates new memory for heap regions and “moves” them there. This region movement should be done when the regions are taken out of the free region list.

- *region locking* – When a region is no longer used, it should be locked, so that any access to it will be caught. This region locking should apply everywhere regions are returned to the free region list, and it should work by removing permissions from the memory where the region was.
- *address mapper* – Since there are places in the code where calculations need to be done on the old region addresses, there should be a mechanism for remapping the new region addresses to the old ones. This mechanism should use hash tables as its foundation, and it should have an easy interface for usage.
- *actual address remapping* – When the address mapper is done, there should be an effort to locate every place where it is needed and put the remapping there.
- *quality of life features* – There should be some “quality of life” changes like printing an error message when the wrong GC is selected, automatically turning off compressed OOPs, etc.

Chapter 5

The Implementation of SanitizeGC

This chapter describes the implementation of the designed changes outlined in the previous chapter. It starts with a description of the implementation platform, and then it focuses on the list in Section 4.4 and goes into technical detail of how and where those changes were implemented. Since there will be mentions of source code files, unless mentioned otherwise, the directory in which the files are located (from the parent directory of this project's code) is `src/hotspot/share/gc/g1/`. Also, every method mentioned has the opening and closing parentheses `()` at the end of its name to clearly distinguish it from variable names, even though the method might take arguments and not have the parentheses empty in the actual code.

5.1 The Implementation Platform

As was already mentioned, the goal of this thesis was to create a proof of concept that I could showcase and argue that the implemented changes would be a great addition to the HotSpot codebase. Because of this, instead of focusing on many platforms and versions, I decided to focus solely on one. The operating system for which the changes will be implemented is Linux, since all my colleagues and I use it on a daily basis, and not all of us have Windows or macOS machines available. There are also architecture-specific parts of code in HotSpot, and for those, I decided to implement the changes for x86-64 for the same reason as above, it is the most common architecture among my colleagues, and not all of us have ARM machines available.

As for the version of Java to extend, I decided on 23, because at the time when we started with the first work on this project (late 2024), it was the most recent version. Currently, the latest Java version is 26, however, I decided that I would rather finish the project on version 23 than rush with rebasing and making the project more unstable. However, the future plan is definitely to eventually rebase to a newer Java version, but that is beyond the time scope of this thesis.

5.1.1 Extension Activation

To activate SanitizeGC, there is a command line option `-XX:+SanitizeGC` (defined in `src/hotspot/share/runtime/globals.hpp`). By default, it is turned off to prevent any unexpected behavior. When turned on, it will set the `SanitizeGC` global variable to `true`. This variable is used almost everywhere where there is a SanitizeGC change, it checks whether it is on, and only if it is, it executes them.

5.2 Moving the Regions

To move the regions, the place where the regions are taken out of the free region list needed to be located first. This place is in the method `HeapRegionManager::allocate_free_region()` inside of `g1HeapRegionManager.cpp`. This method either takes a region with a specific index (if it was specified) or it takes a region from the list's head or tail, depending on the generation of the desired region (old or young), and returns it.

At the end of this method, before returning the region, a new condition was added that checks if `SanitizeGC` is turned on. If so, it calls a new method called `G1HeapRegion::move_free_region()` while logging info about the region. This new method is inside the same class as the heap regions themselves. It is implemented inside the `g1HeapRegion.cpp` file, and it looks like this (with some parts left out):

```
void G1HeapRegion::move_free_region() {
    HeapWord* new_bottom = (HeapWord*) os::reserve_memory_aligned(
        GrainBytes, GrainBytes, false);
    bool set_protection = os::protect_memory(
        (char*) new_bottom, GrainBytes, os::MEM_PROT_RW, true);

    if (!is_moved) {
        bool locked_original = os::protect_memory(
            (char*) _bottom, GrainBytes, os::MEM_PROT_NONE, true);
    }

    // Initialize the maps if it was not done already.
    if (!SanitizeGCRegionMaps::are_initialized) {
        RegionMapEntry::shift_by = LogOfHRGrainBytes;
        SanitizeGCRegionMaps::moved_to_original = new RegionMap();
        SanitizeGCRegionMaps::original_to_moved = new RegionMap();
        SanitizeGCRegionMaps::are_initialized = true;
    }

    // Adding new addresses into the maps and updating old ones.
    if (!is_moved) {
        ... // Described later.
        _is_moved = true;
    } else {
        ... // Described later.
    }

    HeapWord* new_end = new_bottom + GrainWords;
    _bottom = new_bottom;
    _top = new_bottom;
    _end = new_end;
    // Setting parsable bottom and resetting top.
    _parsable_bottom = new_bottom;
    G1CollectedHeap::heap()->concurrent_mark()
        ->reset_top_at_mark_start(this);
}
```

This method does multiple things, firstly, it reserves new memory for the region using the inbuilt method `os::reserve_memory_aligned()`. This method reserves memory and ensures that it starts at an address that is aligned to a given alignment (so that internal address operations, including the address mapper, work correctly). The size of the memory to reserve and the alignment size are the same, and it is defined in the global variable `G1HeapRegion::GrainBytes`, which represents the heap region size. To make sure the memory is aligned, the `os::reserve_memory_aligned()` method uses `mmap()` internally to allocate a bigger chunk than needed, and then it strips the memory to make it aligned.

After that, it sets its protection with `os::protect_memory()` to `os::MEM_PROT_RW`, which means that it will be readable and writable (since this memory will be used right now). Then, it locks the old region's memory by setting its protection to `os::MEM_PROT_NONE` if it is the first time the region was moved. Next, it initializes the address mappers, if they were not already initialized (so they are only initialized when the first region is moved), and adds the new and corresponding old addresses to them (more on that in Section 5.4.1).

After that, it updates the `_bottom`, `_top`, and `_end` addresses of the region that is being moved. The *bottom* address is the address of the first byte of the region, or the starting address, *top* is the currently largest address in the region that is being used (so it will be equal to *bottom*, since the region is not used when in the free region list), and *end* address is the address of the last byte of the region plus one (so when the regions are in sequence next to each other, it is the same address as the next region's *bottom* address). Lastly, it updates some internal variables, and the move is done.

To make debugging easier, it also logs that the region is being moved and it prints its *bottom* and *end* addresses before and after the move in the `HeapRegionManager::allocate_free_region()` method.

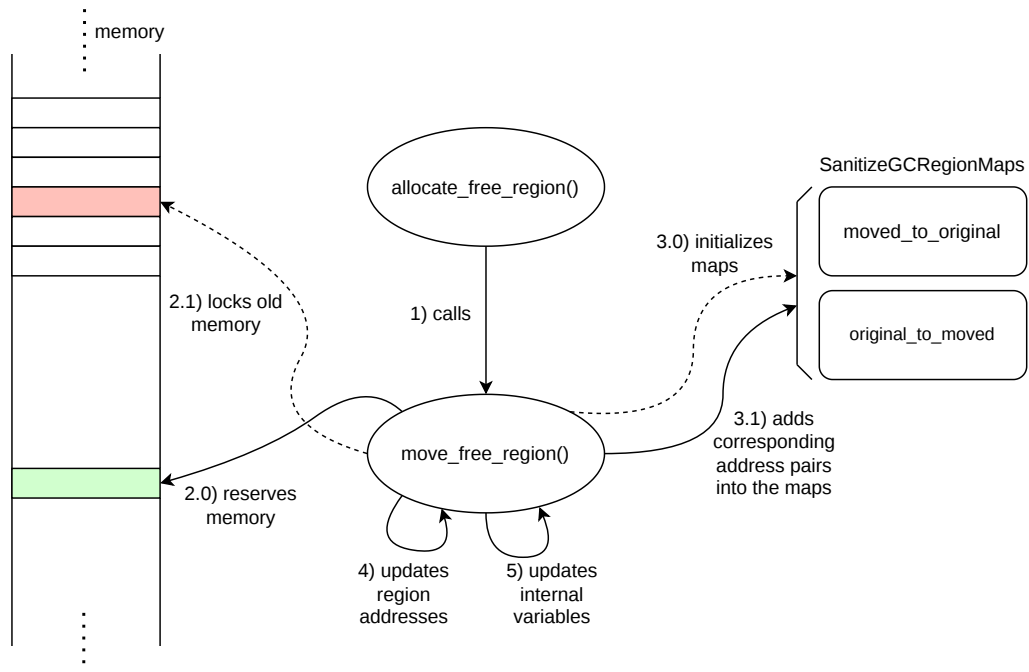


Figure 5.1: A diagram representing the region moving sequence.

Figure 5.1 shows the region moving sequence in order as it happens. Solid arrows represent actions that happen every time, and dashed arrows represent actions that only happen in some cases (locking the region and initializing the maps).

5.3 Locking the Regions

There are multiple places where regions are being locked. The first place was described in the previous section, it is the place where the regions are moved. Here, the locking mechanism is quite simple, it only sets the protection to *NONE*. However, it only does this if the region is being moved for the first time (the first time it is taken out of the free region list), because the initial memory the region started with must not be uncommitted, as that would break some internal assertions.

Other than that, two other places were located where region locking should take place. Both of these are where heap regions are returned to the free region list after all objects are evacuated from it and it is deemed free, and both are in the file `g1CollectedHeap.cpp`.

The first one is inside a method called `G1CollectedHeap::free_region()`. In this method, many things are done, like assertions about the region, resetting some of the region's values, setting the region's type as *free*, rewriting the region's actual memory with garbage bytes (only in debug builds), updating the remembered set tracker, and at the end of the method, the region is returned to the free region list with a `free_list->add_ordered()` call. So, a few things were added to this method with SanitizeGC:

```
...
// Clearing the space is useless with SanitizeGC.
hr->hr_clear(!SanitizeGC /* clear_space */);
_policy->remset_tracker()->update_at_free(hr);

if (SanitizeGC) {
    log_debug(gc, region)("SanitizeGC: freeing region with index
        %d (%p - %p)", hr->hrm_index(), hr->bottom(), hr->end());
    os::uncommit_memory((char*) hr->bottom(), G1HeapRegion::GrainBytes);
    hr->set_uncommitted(true);
}

if (free_list != nullptr) {
    free_list->add_ordered(hr);
}
```

First, when SanitizeGC is on, it does not rewrite the region's memory with garbage, as it is unnecessary even in debug builds, since the memory will be inaccessible. Second, a log is printed with the information about the region that is being returned. Third, the region is uncommitted using the `os::uncommit_memory()` before returning it into the list. As described in Section 4.2.3, this method saves physical memory while protecting it against reads and writes, effectively locking it. And lastly, the region's `_is_uncommitted` variable is updated to `true`.

The second place is in a method called `do_heap_region()` belonging to the `RebuildRegionSetsClosure` class. This method is being called fewer times than the one above, mainly during a full garbage collection, however, it does very similar things:

```

if (r->is_empty()) {
    assert(r->rem_set()->is_empty(),
        "Empty regions should have empty remembered sets.");
    // Add free regions to the free list
    r->set_free();
    if (SanitizeGC) {
        log_debug(gc, region)("SanitizeGC: freeing region with index
            %d (%p - %p)", r->hrm_index(), r->bottom(), r->end());
        os::uncommit_memory((char*) r->bottom(), G1HeapRegion::GrainBytes);
        r->set_uncommitted(true);
    }
    _hrm->insert_into_free_list(r);
} else if (!_free_list_only) {
    ...
}

```

First, a condition checks whether the given region is empty. If it is empty, it checks its remembered set, sets the region type to free, and inserts it into the free region list. Since in the other part of the condition, the region is not inserted into the list, only the first part was extended. If SanitizeGC is on, before the insertion, it prints a log describing the region, uncommits its memory, and sets its `_is_uncommitted` variable to `true`.

The `_is_uncommitted` is then used in the file `g1HeapRegion.inline.hpp` in a `G1HeapRegion::reset_after_full_gc_common()` method. After a full garbage collection is done, it rewrites (or, as the code says, “mangles”) unused region memory with garbage bytes here in debug builds. Since this is unnecessary with SanitizeGC, it checks if the region was uncommitted, and if so, it is skipped.

The `_is_uncommitted` variable is then reset to `false` when the region is taken out of the list and moved again.

5.4 Address Conversion

In this section, the actual address conversion from moved to non-moved (and vice-versa) is described. The focus will first be on the actual implementation of the address mappers as they were designed in Section 4.2.2, and then, there is a subsection about how the actual places that need remapping were found, with an example of how the address remappers were used there.

5.4.1 Address Mappers

The mappers’ actual implementation is split across multiple files. Firstly, there are the `sanitizeGCHeapRegionMap.hpp` and `sanitizeGCHeapRegionMap.cpp` files, in which the mechanism that works with the underlying hash tables is implemented. Then there are the `sanitizeAddressMapper.hpp` and `sanitizeAddressMapper.cpp` files, which serve as wrappers around the tables, so that the interface is simple to use everywhere in the code.

`sanitizeGCHeapRegionMap`

In the `sanitizeGCHeapRegionMap.hpp` file, the *region map* structure is created. The main goal of the region map is to store the starting (bottom) addresses of regions and their remapped counterpart and offer methods for working with it. The region map is based

on `ConcurrentHash Table`, which is an internal HotSpot hash table implementation that supports concurrency (meaning that inserting and retrieving elements should be safe even with multiple threads).

This file defines multiple classes related to the region map. The first one is `RegionMap Entry`, and it contains the key and value parameters. Those parameters are of type `const void*`, and they represent the starting address of a region and its remapped counterpart.

In the same class, there is also a static parameter called `shift_by` and a static helper method `shift_address()`. The `shift_by` value is set in the region remapping method when the maps are initialized, and it is set to the global value `G1HeapRegion::LogOfHRGrain Bytes`. This value is calculated as a $\log_2(n)$, where n is the size of a heap region in bytes. So if each heap region has, for example, 16 MiB, the value would be $\log_2(16\,777\,216) = 24$. This value is very important because it represents the number of (lower) bits in an address that address the memory within a single region. This is used in the `shift_address()` method. This method takes an address pointing to a region and shifts it right by `shift_by` bytes, meaning that only the part of the address shared by all addresses in that region remains. This shifted address is then converted to `uintx` and returned (as shown in Figure 5.2).

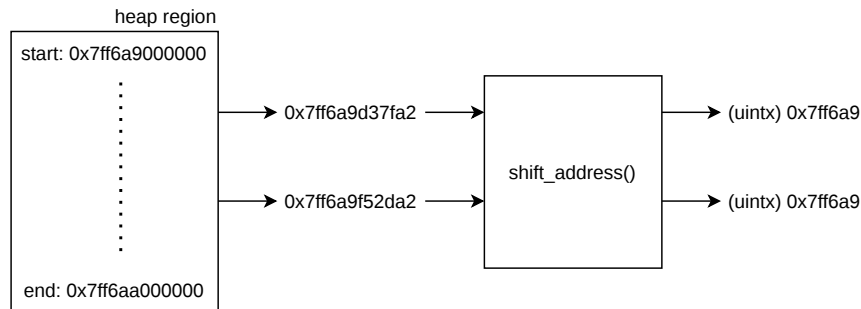


Figure 5.2: For any address from a single heap region, the output of `shift_address()` is the same. The `shift_by` value is 24 in this case.

Next class in the file is `RegionMapConfig`, and it contains three methods: `get_hash()`, which just calls the `shift_address()` method from the previous class to create a “hash” of an address (since there should not be more entries for a single region in a single map, this “hashing mechanism” is enough in this case), `allocate_node()`, which allocates memory for the map entry on the heap, and `free_node()`, which frees the memory for an entry.

Next class is called `RegionMap`, and it is the most important class of this file, as it is the actual class of the region map. Firstly, there are two helper subclasses defined inside this class. These subclasses need to be defined, so that the operations on `ConcurrentHashTable` work correctly. The first one is `RegionMapLookUp`, and it is used for looking up entries in the map based on a given key/address. The second one is `RegionMapGet`, and it is used for retrieving the value corresponding to the key. After that, the actual constructor of this class is defined, which just creates a new `ConcurrentHashTable` and saves it into the `_table` parameter.

Other than that, there are a few methods for actually working with the hash table in `_table`. The first of those is `insert()`, which gets the key&value pair as an argument and inserts it into it, if an entry with the same key is not there already. The next one is `update()`, which also takes a key&value pair. It removes an entry from the table with the

given key, then inserts the key again with the new value this time. After that, there is a `remove()` method, which just removes the entry with a given key.

Next continues a method called `remap_address()`, which is probably the most important method in this file, as it takes an address as a parameter and remaps it. Here is the whole code of it:

```
const void* remap_address(const void* addr) {
    if (SanitizeGC) {
        RegionMapLookup lookup(addr);
        RegionMapGet rmg;

        bool found = _table.get(Thread::current(), lookup, rmg);
        if (found) {
            // remap the given address to the one which was found
            const uintptr_t mask_bits =
                ((uintptr_t)1 << RegionMapEntry::shift_by) - 1;
            const uintptr_t low_bits_to_add = (uintptr_t)addr & mask_bits;

            return (const void*)
                ((uintptr_t)rmg.get_entry_value() | low_bits_to_add);
        }
    }

    // no matching entry was found
    return nullptr;
}
```

As can be seen, the method first checks whether `SanitizeGC` is enabled and if so, it allocates `RegionMapLookup` (with the given address in `addr` serving as a key to look for), and `RegionMapGet` objects. After that, it tries to retrieve a corresponding element (corresponding remapped region starting address) from the hash table using the `_table.get()` call. If an element was found in the table, it first creates a bit mask by shifting “1” left by `shift_by` bits and subtracting “1” from the result, and then it uses this mask to get the lower bits from the given `addr`. After that, it uses the table entry it found (which will always have the lower bits set to zeroes, as only starting addresses of a region are saved in the table), adds those lower bits to it, and returns the result. If `SanitizeGC` is not tuned on or no element corresponding to the given key was found, the method returns `nullptr`.

Another method in this class is called `remap_end_address()`, and it is used in special situations where the end address of a region needs to be remapped. The problem with these end addresses is illustrated in Figure 5.3. When the regions are right next to each other (the state in which they originally are), the end address of a region is the same address as the start (bottom) address of the region right after it. However, when the regions are moved, they do not end up right next to each other, but they are scattered throughout the memory, so the end and start addresses of two subsequent regions are no longer shared.

This causes problems when an end address from the original region memory space needs to be remapped. Since the `remap_address()` method does not have context about the type of address that is remapped, it would take this address and remap it into the starting address of the next region, because that exact value is stored in the map (only starting addresses



Figure 5.3: A figure showing that a region’s *end* address is shared with the next region’s *bottom* address when they are next to each other.

of regions are stored in the map). To mitigate this, `remap_end_address()` exists with this code:

```
const void* remap_end_address(const void* addr) {
    const void* result = nullptr;
    if (SanitizeGC) {
        uintptr_t converted_addr = (uintptr_t)addr;
        const void* remapped =
            remap_address((const void*)(converted_addr - 1));
        if (remapped != nullptr) {
            result = (const void*)((uintptr_t)remapped + 1);
        }
    }

    return result;
}
```

As can be seen, the trick in this method is very simple, it subtracts “1” from the potentially end address to make sure it will not be the end address but rather at the worst case the one right before end, then it remaps this subtracted address using the `remap_address()` described above and it adds the “1” back to the result before returning it. This method is very situational, and it is used in cases where region end address remapping is being done.

The last thing in the file is the `SanitizeGCRegionMaps` class, which just contains declarations for two static region maps and a boolean to check whether they were initialized (they are initialized in `g1HeapRegion.cpp`). The maps are `moved_to_original` for mapping region’s new/moved addresses to their original counterpart before it was moved, and

`original_to_moved` for mapping the region's original addresses it started with to their new addresses after the region was moved.

There is also `sanitizeGCHeapRegionMap.cpp`, but it contains nothing interesting, just an initialization of placeholder values for static parameters.

`sanitizeAddressMapper`

In the `sanitizeAddressMapper.hpp` file, there is a definition of a class called `SanitizeGCMapper`. This class contains multiple methods, and as a whole, it serves as a wrapper for the maps. First, there are four method declarations:

- `mapNewAddrToOriginalAddrImpl()`
- `mapNewEdgeAddrToOriginalAddrImpl()`
- `mapOriginalAddrToNewAddrImpl()`
- `mapOriginalEdgeAddrToNewAddrImpl()`

These are actually implemented in `sanitizeAddressMapper.cpp` and they all work very similarly. Here is an example code of one of those:

```
const void* SanitizeGCMapper::mapNewAddrToOriginalAddrImpl(
    const void* newAddr) {
    if (SanitizeGCRegionMaps::are_initialized) {
        const void* remapped = SanitizeGCRegionMaps::moved_to_original
            ->remap_address(newAddr);
        if (remapped != nullptr) {
            return remapped;
        }
    }

    return newAddr;
}
```

The remaining methods work in the same way. All of these take an address as an argument, and, if the region maps were initialized, try to remap the address using one of the map methods described in the section above. The “new address to original address” methods use the `moved_to_original` map, and “original address to new address” use the `original_to_moved` map. If the method has “edge” in its name, it uses the `remap_end_address()` method for the remapping, otherwise, the standard `remap_address()` is used. It is also worth noting that if an address is not found in the map (`remap_address()` returns `nullptr`), no remapping is done and the given address is returned as is.

Other than these methods, the `SanitizeGCMapper` class also contains four other method definitions in the header file:

- `mapNewAddrToOriginalAddr()`
- `mapNewEdgeAddrToOriginalAddr()`
- `mapOriginalAddrToNewAddr()`
- `mapOriginalEdgeAddrToNewAddr()`

These are all template methods, meaning that they take and work with a generic type, as opposed to the four `Impl` methods above, that take and return `const void*`. They also only take a reference to an address as an argument and do the remapping in place (by calling one of the `Impl` methods and storing the casted result into the argument), meaning that they don't return anything, but still change the given address. This makes them very versatile and allows them to be used almost everywhere in code, even in places where addresses are represented by some custom types. Here is an example code of one of those methods:

```
template <typename T> static inline void mapNewAddrToOriginalAddr(
    T &addr) {
    addr = (T) mapNewAddrToOriginalAddrImpl(addr);
}
```

Other than that, there is a macro definition (`MacroAssemblerMapNewAddrToOriginalAddr()`) in this file for remapping a new address to its original counterpart. This macro is used in only one file, which generates assembly code to speed things up, but still needs addresses to be remapped. This macro uses other special assembly macros to call the remapping method and return the result into a special register, where it is later used.

5.4.2 Adding New Addresses to the Maps

New addresses are inserted into the maps in the `G1HeapRegion::move_free_region()` method that is described above. If a region is moved for the first time, its bottom address and the new bottom address (after the move) are simply inserted into both maps like this:

```
SanitizeGCRegionMaps::moved_to_original->insert(new_bottom, _bottom);
SanitizeGCRegionMaps::original_to_moved->insert(_bottom, new_bottom);
```

If the region was already moved before and is moved again, this code gets triggered:

```
HeapWord* original_bottom = (HeapWord*)
    SanitizeGCRegionMaps::moved_to_original->remap_address(_bottom);
SanitizeGCRegionMaps::moved_to_original->
    insert(new_bottom, original_bottom);
SanitizeGCRegionMaps::original_to_moved->
    update(original_bottom, new_bottom);
```

As can be seen, the first thing that happens is that the original bottom address is found using the `remap_address()` method (because the region was already moved, it is not as straightforward to get the original bottom address as in the first case, it is saved in the map, however). After that, the new address with the original bottom is inserted into the `moved_to_original` map, and the `original_to_moved` is updated to have the new address instead of the older one. As can be seen, the old address pair, which is now invalid, is not removed from the `original_to_moved` map. This is because of the Snapshot-At-The-Beginning algorithm. In some edge cases, a region might be moved after the snapshot of the heap is taken. For this, the old pairs still need to be in the `moved_to_original` map, so that the remapping is done even in this case. If the pair were removed from the map, it would fail.

5.4.3 Address Remapping

The actual locations where address remapping needs to occur are scattered throughout the code. As far as implementation goes, finding those locations was the hardest part, as there is no easy way to do it. In the end, more than 30 locations were identified where remapping is needed. Those locations were identified by one of two methods: The first method was static analysis of the code, finding potential remapping locations, debugging this place, analyzing the behavior, and seeing whether it really needs remapping or not. While this is the ideal analysis option, it was really time-consuming, because only the G1 code is made of about 250 files and has more than 30000 lines of code (and also other places than G1 were also touched, such as shared card table code, adding to the LoC total that needed to be analyzed).

Because finding these locations solely through static analysis was almost impossible, it was combined with an approach similar to *test-driven development*. Throughout the development cycle, multiple tests were created to stress the G1 GC and expose potential missing address remappings. Most of the tests would end up in a crash. These crashes were analyzed and debugged, and the cause was eventually identified and patched.

Simple crashes were identified quite quickly and easily, but when the crash was more complex, the process of finding the cause was very hard and time-consuming. Great help in this phase was a tool called **rr**¹. The **rr** is a C/C++ debugging tool for Linux that enhances **gdb** – it allows the user to record a program’s run and then deterministically debug the recording as many times as the user likes, allowing them to set different breakpoints or watchpoints each replay and even reverse-execute the program to a certain point [32].

Because garbage collection is nondeterministic by nature and might be done differently in each run of the same test, **rr** was sometimes used to make the debugging easier, as the replay is the same each time.

Below is an example of a method that was located as a candidate for remapping (from the `g1CollectedHeap.inline.hpp` file):

```
inline uint G1CollectedHeap::addr_to_region(const void* addr) const {
    SanitizeGCMapper::mapNewAddrToOriginalAddr(addr);
    ...
    return (uint)(pointer_delta(addr, reserved().start(), sizeof(uint8_t)) >>
        G1HeapRegion::LogOfHRGrainBytes);
}
```

The method calculates an index of a region based on a given address pointing to it. It does this by calculating the pointer delta between the given `addr` and `reserved().start()` and bit shifting the result. It does not take a lot to see that if the given `addr` was pointing somewhere else than to the expected original memory of the regions, the calculation would return an unexpected value. That is why `SanitizeGCMapper::mapNewAddrToOriginalAddr()` is called before that to remap the address to its original state, making the calculation correct again.

There are also files in HotSpot that generate assembly code using special mechanisms and macros for critical parts of the code to speed things up (but at the cost of platform-specificity). One of these files is `g1BarrierSetAssembler_x86.cpp`. It creates assembly code that is later run. The problem is that there is a place in this file that also needs address remapping. Finding this took a lot of time, since it included debugging assembly

¹<https://rr-project.org/>

code and finding function calls/known structures inside it, to identify the places in the file that generated the crashing assembly. However, after a while, the places were identified, and the macro mentioned in the section above is used there to remap the addresses.

5.4.4 The Whole Process

The Figure 5.4 shows the whole address remapping process on an example method (`G1CollectedHeap::addr_to_region()`). The method holds an address in a moved region (marked by *new* in the diagram), and it needs it to be remapped to a corresponding address in the original region (marked by *old*). As can be seen, it calls `mapNewAddrToOriginalAddr()` for the remapping, which in turn calls `mapNewAddrToOriginalAddrImpl()`. The actual remapping is done in `remap_address()`, which searches for the corresponding address in the `moved_to_original` map and adds the lower bits to it. After that, the result is propagated back to the `addr_to_region()` method, where it replaces the previous address.

The diagram also shows the `move_free_region()` method, which adds new address pairs to the maps using `insert()` when a region is moved.

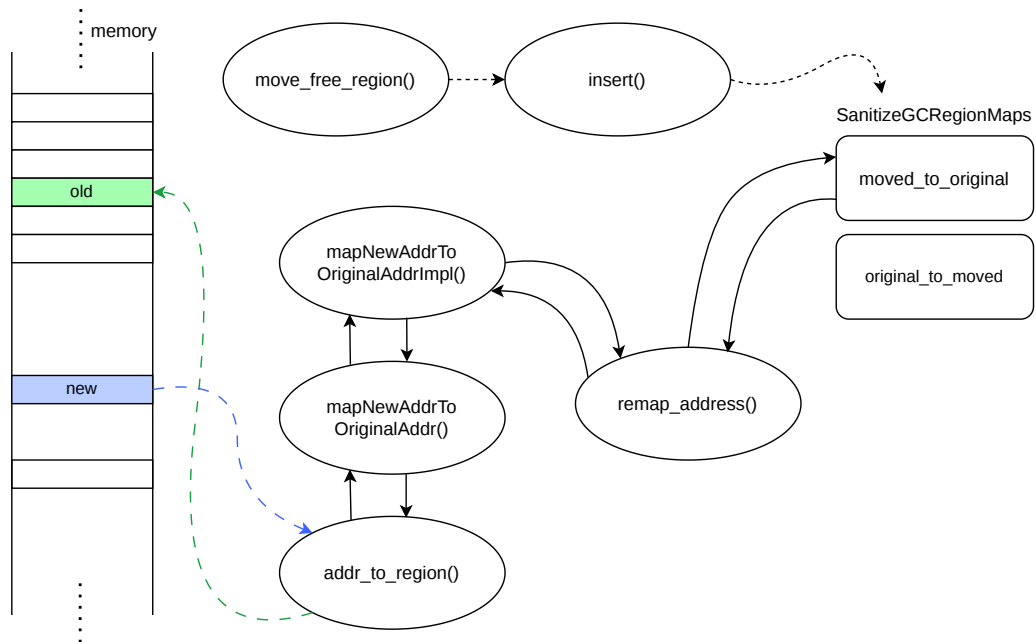


Figure 5.4: A diagram representing a whole address remapping process.

5.5 Quality of Life Changes

In addition to the main changes, some “quality of life” changes were implemented to make working with `SanitizeGC` easier. One of those changes is a method called `printMemoryRegionMap()` implemented inside `g1HeapRegion.cpp`. This method iterates through all memory regions and prints info about each one, including its bottom and end addresses, its index, whether it was moved (and, if so, the original bottom and end addresses), its current top, and the generation this region belongs to. This method is especially useful

while debugging, it can be called directly in `gdb` during a debugging session by writing `call printMemoryRegionMap()`.

Other than that, small changes were made here or there. These include automatically turning off compressed OOPs (and writing an error message if the user turns them on manually together with `SanitizeGC`), writing a message if a garbage collector other than G1 was selected with `SanitizeGC` turned on, or writing a simple error message when the JVM catches an invalid read/write operation, telling the user that it might have been because of the extension catching a dangling pointer. Also, some asserts and conditions were changed to account for the `SanitizeGC` changes.

Chapter 6

Testing the SanitizeGC Changes

This chapter focuses on testing conducted for the SanitizeGC project. The project was first tested on a custom test suite created during the implementation phase, and then the G1 test suite, which was already present in HotSpot’s code, was used to see how the project stands. Since the custom test suite has many tests that stress the GC, it was also used as a benchmark to measure the difference between having SanitizeGC on or off. After that, the project was tested on two real-life applications to see how it performs in real conditions. Lastly, two special scenarios were created to introduce bugs into the code that SanitizeGC should be able to detect, demonstrating that it is a useful tool.

6.1 Different Build Types

Before describing the actual testing, it is important to mention the different build types the JDK can have. The build types differ in the level of debug information they produce. There are four main types: **release**, **optimized**, **fastdebug**, and **slowdebug** [18]. In this project, the focus was on two of those, the **release** and **slowdebug**. The **release** does not contain debug symbols and does not perform any assertions, making it the fastest and ideal for distributing binaries. On the other hand, **slowdebug** contains debug symbols, making it ideal for cases where debugging of the JVM itself is needed. However, it is also significantly slower, making it not ideal for day-to-day use.

6.2 Functional Tests

The functionality testing is described in this section. The focus is on proving that SanitizeGC works as intended through a set of SanitizeGC and G1 tests, benchmarking it to see how it holds up, and running real-life applications with SanitizeGC to further demonstrate its usability.

6.2.1 Test suites

As already mentioned, the main functional testing was done on two test suites, the SanitizeGC test suite, created during development, and the G1 test suite, already present in HotSpot’s codebase. The tests in both of these are written in Java, and they are so-called *jtr* tests. The *jtr* tool is a test harness tool that is used in the JDK for running the tests and seeing their results [21].

The SanitizeGC test suite is located in the `test/hotspot/jtreg/gc/SanitizeGC` directory of the source code, and it currently contains 30 tests. 12 of those tests are designed to stress the GC in various ways, such as allocating millions of short-lived objects and performing operations on them, or creating large strings or byte arrays. These are located in a `benchmarks` subdirectory, and they are later used for the performance comparison. There are also 4 tests that were designed to fail. They mainly test wrong combinations of arguments or early exit from the JVM (there is also one GUI test included in this category, it might run correctly depending on the setup). These are located inside a `shouldfail` subdirectory, and they require manual inspection when running the test suite. Apart from these two special categories, there are 14 more tests that do various things, such as running SanitizeGC with a very small or very large heap size, testing subprocesses, exception catching, trying to create a simple HTTP server, testing simple multithreading, etc. The goal of most of those tests was to simply run the whole program correctly without crashing and finish with an exit code 0.

The G1 test suite contains 73 tests and is located in `test/hotspot/jtreg/gc/g1`. The tests here are designed to thoroughly check the G1 garbage collector and its abilities. Since these were not designed for SanitizeGC, it needed to be manually enabled with jtreg by using this argument: `-javaoption:-XX:+SanitizeGC`, which adds this option to every test the jtreg runs.

Both of those test suites were run with both `release` and `slowdebug` builds. The results of SanitizeGC test suite are in Table 6.1. As can be seen, only three tests failed and those are the ones that are designed to fail (the GUI test was successful in this case). Upon manual inspection, they failed correctly in both builds. This was, however, expected, since those those tests were run during the SanitizeGC development and every failure in them was heavily debugged and fixed.

	release	slowdebug
PASSED	27	27
FAILED	3	3

Table 6.1: The results of the SanitizeGC test suite.

The situation is more complicated in the G1 test suite, the results are in Table 6.2. As can be seen, in both builds, the number of failed tests is the same (as well as the number of tests that had errors), the number of passed tests is slightly larger with the `slowdebug` build, but that is because of the fact that more tests were run (some tests are only meant to be run with `slowdebug` builds).

	release	slowdebug
PASSED	41	47
FAILED	20	20
ERROR	1	1
NOT RUN	11	5

Table 6.2: The results of the G1 test suite.

The one test that had an error (in both builds) is irrelevant in this case, as the test also has an error with the SanitizeGC turned off. For the `release` build, 14 tests out of the 20 failed because they work with humongous regions/objects (most of those contain *humongous* in their name or mention a method that works with humongous regions in the

crash message), meaning that these failures were expected and not relevant for the project in this stage. Also, 1 of those 6 non-humongous failures occurs with SanitizeGC turned off, so in total, there are 5 relevant failures.

For the `slowdebug` test failures, a comparison was created with the `release` failures. From this comparison, it was found that there are two new unique failures in `slowdebug` (and since there is the same number of failed tests in both builds, it means that there are two unique failures with the `release` build). One of those failures is related to compressed OOPs, the test tries to turn them on automatically, which is forbidden with SanitizeGC.

6.2.2 Known Failures

So, in total, 6 unique relevant (excluding humongous failures, compressed OOPs failures, and failures that are also present without SanitizeGC) failures were found in the G1 test suite:

- `TestConcurrentSystemGC`
- `TestShrinkAuxiliaryDataRunner`
- `TestVerificationInConcurrentCycle#id1`
- `ihop/TestIHOPergo`
- `ihop/TestIHOPStatic`
- `plab/TestPLABEvacuationFailure`

Checking the logs of those failures reveals that the `TestConcurrentSystemGC`, `TestShrinkAuxiliaryDataRunner`, `TestVerificationInConcurrentCycle#id1`, and `plab/TestPLABEvacuationFailure` tests failed due to the same error. The problematic frame that invoked a segmentation fault came from the native standard C library `libc.so`, and it is this: `__memcpy_avx_unaligned_erms`. This is currently the biggest debugging effort in this project, but it seems to be a very difficult bug to fix, as the stack trace of this seems very ambiguous. Since `memcpy` copies memory, the crash might also indicate a legitimate use-after-free vulnerability, so extra effort is needed to debug this to either prove or disprove this.

The `slowdebug` run of the `plab/TestPLABEvacuationFailure` has a bit more information, it failed on a different thing than the `release` run. The problematic frame here is from the JVM itself, and it is `oopDesc::klass()`, so the attention while investigating this will be on this method. Also, there were successful attempts to reproduce this failure outside of these tests. The failure can be forced in some special cases, where the maximal memory of the JVM is set to be very small and a lot of large, but not humongous objects are allocated (hundreds of thousands). Making the memory larger makes this error disappear.

Apart from that, the `ihop/TestIHOPergo` (failed with `slowdebug` only) test failed an assert inside of the `pointer_delta()` method. The exact cause has not yet been found, however, since the test ran correctly with the `release` build, it looks like the assert might not be important with SanitizeGC. The last failure, `ihop/TestIHOPStatic`, failed on the `G1ParScanThreadState::trim_queue_to_threshold()` method with `release` build, but it failed an assert inside of the `CompressedKlassPointers::decode_not_null()` method with `slowdebug`. Both of those failures are currently under investigation, and fixing them is a priority. However, it is slow and not an easy process, as the HotSpot's codebase is very complex, with tens of thousands of lines of code.

6.2.3 Benchmarks and SanitizeGC Performance

As already mentioned, for benchmarking, the 12 “stress” tests from the SanitizeGC test suite were used:

- **ArrayAndListBenchmark** – Compares the speed between a primitive integer array and an `ArrayList` of integers.
- **ByteStressTest** – Allocates a lot of byte arrays in a loop.
- **CalculationsBenchmark** – Allocates a lot of objects and does calculations (XOR, division, etc.) on them.
- **CombinedBenchmark** – Allocates a lot of objects, doubles, and strings in a loop.
- **ConcurrencyBenchmark** – Benchmarks working with multiple threads, incrementing a single integer (comparing `synchronized` block and `AtomicInteger`).
- **FullGCTest** – Allocates byte arrays in a loop with a full GC triggered in the middle of the run manually.
- **ObjectStressTest** – Allocates millions of objects of different sizes in a loop.
- **PeriodicGCBenchmark** – Allocates objects in a loop with a GC triggered every 100 milliseconds.
- **SimpleInterpretedBenchmark** – Allocates byte arrays in a loop and runs in interpreted mode (no JIT).
- **SmallObjectsBenchmark** – Allocates 250 million small objects in a loop.
- **StringBuilderBenchmark** – Creates long strings using a `StringBuilder`.
- **StringStressTest** – Creates a lot of strings in a loop.

A custom Python script was used to run every test with five different maximum heap sizes (100 MB, 200 MB, 400 MB, 800 MB, and 1600 MB), ten times in total for each variation (so there were 600 runs in total). These benchmarks in this script are run three times in total, once with the SanitizeGC extension enabled (**release** build, as measuring **slowdebug** would be counterintuitive), once without SanitizeGC, but with compressed OOPs turned off, and once with normal JVM settings (no SanitizeGC and compressed OOPs on). The script measures the elapsed time and resident set size (RSS) using the `/usr/bin/time -v` command, and parsing its output. The results from this are saved in CSV files. A second Python script was used to generate graphs out of these results.

The final benchmarks were run on a machine with an Intel Core Ultra 7 CPU, 64 GB of RAM, and Fedora Linux as its operating system. Figures 6.1 and 6.2 show the average time usage comparison for each test and the maximum heap size. Figures 6.3 and 6.4 show the average memory (RSS) comparison for each test and the maximum heap size. The figures are located at the end of this chapter to avoid formatting errors.

As shown in the graphs, the time difference between *vanilla* JVM (SanitizeGC turned off and compressed OOPs not disabled) and JVM with compressed OOPs disabled is very minimal, except for the **ArrayAndListBenchmark** and **SmallObjectsBenchmark** tests, where the difference can be seen (it is probably due to the added overhead of larger object headers

with small objects). The no compressed OOPs variant was added to see whether implementing compressed OOPs into SanitizeGC would make it faster, however, the difference is not that significant. The memory usage difference is also very minimal except for the `ArrayAndListBenchmark` test.

As can also be seen, the largest time difference between SanitizeGC on and off occurs in tests that are extra heavy on the garbage collector, allocating many (up to a few million) short-lived objects or very large byte arrays. These include tests like `ByteStressTest`, `FullGCTest`, `ObjectStressTest`, `StringStressTest`, etc. On the other hand, in tests that focus more on different operations than solely on object allocation (like `CalculationsBenchmark` or `ConcurrencyBenchmark`), the time difference is smaller, almost negligible.

As for SanitizeGC's memory usage, it is mostly on par with vanilla JVM, except at the largest heap settings, where it shows a slight peak in most tests compared to the other two variants. These peaks are again present in the tests, where a lot of objects are allocated, in tests like the `CalculationsBenchmark`, the peak is not present at all, and SanitizeGC even consumes less memory than the rest. In the `ArrayAndListBenchmark` test, it consumes almost the same amount of memory as the no compressed OOPs variant, the difference with vanilla JVM is the largest here.

In summary, SanitizeGC was, on average, **4.5 times slower** than a normal JVM, with a maximum measured difference of 6.4 times slower. It used, on average, **1.1 times more memory** than a normal JVM, with a maximum measured difference of 2.1 times more memory. Compared to the JVM with compressed OOPs turned off, it was, on average, 4.2 times slower and used almost the same amount of memory.

6.2.4 Running SanitizeGC With Real-Life Applications

To further test SanitizeGC's functionality, it was tested (the `release` build) on a few different “real-life” applications. Those applications were the H2 SQL database engine¹, and three Quarkus Quickstarts².

The H2 Database Engine can be downloaded as a single precompiled JAR file. After downloading it, it was run with the SanitizeGC JVM with SanitizeGC turned on. Upon starting, the database opens a web interface, in which there is a field for SQL queries. A few operations were tested inside it. These include creating new tables, adding thousands of values to them, and selecting data from them based on some limitations. Everything tested worked correctly, and there was no significant time loss felt during the operations.

Quarkus Quickstarts is a collection of small projects meant as a quick starting point for the Quarkus framework. While Quarkus is optimized to run natively (using GraalVM or Mandrel to compile it into native code), it can also be compiled into bytecode for standard JVM execution [31]. Out of these Quickstarts, the general `getting-started` was first used. It creates a simple CRUD service with a few REST endpoints. The endpoints were tested, and everything worked correctly. Also, this service's start time was almost the same with or without SanitizeGC. Apart from this, it was also tested on the `microprofile-health-quickstart` and `websockets-quickstart` Quickstarts with the same results. Everything worked correctly, and the start time was almost the same as without SanitizeGC. This was a great sign because Quarkus is quite a complex framework, so seeing that SanitizeGC runs it correctly means that it is usable even for larger applications, not just simple one-file tests.

¹<https://h2database.com/html/main.html>

²<https://github.com/quarkusio/quarkus-quickstarts>

From these real-world application examples, it can be seen that SanitizeGC works correctly in most normal cases, with minimal execution time loss. In the next section, there is a description of bug seeding efforts that were done to provide proof that SanitizeGC really catches wrong memory accesses.

6.3 Bug Seeding

To prove the usefulness of SanitizeGC, a so-called *bug seeding* was done. In this bug seeding, memory bugs were introduced deliberately into the code in the hopes that SanitizeGC would successfully find them. Two separate bug seeding efforts were done. The first bug seeding introduced a bug directly in HotSpot's codebase, and the second used a *Java Native Interface* (JNI) method to cause a faulty memory access. JNI is an interface that enables Java code running in a JVM to interact with applications or libraries written in other programming languages like C/C++ [27].

6.3.1 HotSpot Bug Seeding

To introduce a memory bug into the HotSpot's code, a mechanism called *handle* was misused. Handles are provided in the code to safely store pointers to objects. The problem with pointers to objects is that if a method performs operations on an object and, during those operations, a GC is triggered, the object might be moved, leaving the pointer pointing to the old, now invalid, location. Handles prevent this by saving the pointer and updating it when a GC moves the object. After that, the pointer to the object can be accessed using its handle, and the handle itself can be passed around like a standard variable.

So, for our bug seeding, the goal was to find code that uses these handles without any additional checks or assertions, and replace the handle with the raw pointer it holds. This would simulate a very possible error that a programmer might make while developing the JVM (and making this error was one of the inspirations for coming up with this project). If the object of this raw pointer is moved, the pointer might stay pointing to a wrong place, to a memory that should no longer be accessed.

After looking through the code for a while, a place for this bug seeding was located. It is in the `java_lang_Class::allocate_mirror()` method inside of the `src/hotspot/share/classfile/javaClasses.cpp` file. Right at the start of this method, it creates an object pointer (oop means "ordinary object pointer") and its handle:

```
// Allocate mirror (java.lang.Class instance)
oop mirror_oop = InstanceMirrorKlass::cast(vmClasses::Class_klass())->
    allocate_instance(k, CHECK);
mirror = Handle(THREAD, mirror_oop);
```

Later, it uses the `mirror` handle many times to work with that object pointer. However, the important line is this one, which calls a method to save a value (its field count) inside the object:

```
set_static_oop_field_count(mirror(),
    mk->compute_static_oop_field_count(mirror()));
```

To exploit this and introduce a bug, here is how the method was edited:

```

// Allocate mirror (java.lang.Class instance)
oop mirror_oop = InstanceMirrorKlass::cast(vmClasses::Class_klass())->
    allocate_instance(k, CHECK);
mirror = Handle(THREAD, mirror_oop);

// force a GC by allocating a lot of unused data
printf("SANITIZEGC BUGSEEDING: Trying to trigger a GC now.\n");
for (int i = 0; i < 250; i++) {
    typeArrayOop unused_data = oopFactory::new_byteArray(300, THREAD);
}

...

printf("SANITIZEGC BUGSEEDING: the value should be: %p and is %p.\n",
    mirror(), mirror_oop);

// replaced first mirror() call with mirror_oop
set_static_oop_field_count(mirror_oop,
    mk->compute_static_oop_field_count(mirror()));

```

I added several things to the code. First, a loop allocates byte arrays to fill the heap with unused data. The goal is to trigger the garbage collector right after creating the handle, hoping it moves the original `mirror_oop`. The GC might trigger here even without this (for example, due to periodic collection or the heap filling up), but this loop helps make the example bug seeding reproducible.

The next change is a statement that prints information about the expected pointer value from the handle (the correct one) and the value of the original pointer. If the GC happened and it moved the original object, these two values would be different, which is the situation this bug seeding is trying to create.

The last and most important change is the last statement that sets the field count. The first argument of this call was `mirror()`, which uses the handle, but now, it is set to the original object pointer (`mirror_oop`), meaning that the field count will be written to the original object's location, wherever that is. This simulates an error that a programmer might have created by forgetting to use the handle in this case. While this exact case might seem theoretical, new HotSpot code is being written every day, and forgetting to use a handle when working with object pointers is a real possibility, which SanitizeGC might help prevent early on.

For the actual testing of this, the edited HotSpot was first compiled into a `slowdebug` build of JVM, and it was run without SanitizeGC turned on on a simple Java program. Without SanitizeGC, the program exited without crashing, with an exit code of zero. The output was saved and later inspected, and this line was found:

```

SANITIZEGC BUGSEEDING: the value should be: 0x4570005d0
and is 0x457fa9140.

```

As can be seen, the two values differ, indicating that the field count value was really written to a different location than it should have been. However, this did not cause any crashes or errors. In more complex cases, something similar to this might be exploited to do harm and cause unexpected behavior.

When this edited JVM was run with SanitizeGC turned on, it actually crashed, printed a segmentation fault error message, and saved additional information to a crash log. The last line the JVM printed was this, meaning that the JVM did not continue after that point, and it stopped immediately:

```
SANITIZEGC BUGSEEDING: the value should be: 0x7fd00b000928
and is 0x7fd068f9bc68.
```

The crash log also shows the correct stack trace, leading to the spot that was edited to contain a bug (even with the exact line number, `javaClasses.cpp:1002` in this case):

```
V [libjvm.so+0x5404e0] oopDesc::int_field_put(...)+0x26
  (oop.inline.hpp:240)
V [libjvm.so+0xa59866] java_lang_Class::set_static_oop_field_count(...)
  +0x86 (javaClasses.cpp:1198)
V [libjvm.so+0xa589de] java_lang_Class::allocate_mirror(...)+0x1d4
  (javaClasses.cpp:1002)
V [libjvm.so+0xa58e76] java_lang_Class::create_mirror(...)+0x168
  (javaClasses.cpp:1063)
```

Also, there is this line in the crash log indicating that this failure might have been caught using SanitizeGC:

```
SanitizeGC: This was probably caused by trying to dereference a pointer
into locked memory.
```

To sum up, in this bug seeding, a bug was introduced into the HotSpot's code. Without SanitizeGC, the bug was not detected (even though our logging showed that there was really a wrong access to a memory in one case). With SanitizeGC, this wrong access was detected immediately, crashing the program and creating a helpful crash log with a stack trace leading to the exact location where the bug was introduced. This shows that SanitizeGC has the potential to be helpful during JVM development or when debugging bugs in it.

6.3.2 JNI Bug Seeding

While the Java Native Interface is a very powerful tool, it also brings a lot of possible mistakes a programmer might make that can lead to memory vulnerabilities. For this bug seeding, a great thanks goes to my colleague Thomas Stüfe, who helped with creating it. The bug seeding in this section is based on his work³.

The bug seeding has two parts, the Java part and the native part (C code in this case). It is important to note that no part of the HotSpot's code was touched in this bug seeding (except for the SanitizeGC changes, of course), it is only on the user side.

The Java part is in a file called `BadAccess.java`, and it contains this code:

```
package de.stuefe.repros;

public class BadAccess {
    static {
        System.loadLibrary("badaccess");
    }
}
```

³<https://github.com/tstuefe/jni-misuse-crash-example>

```

private static native void badAccess(byte[] array);

public static void main(String[] args) throws Exception {
    // fill an array with ones
    byte[] ar = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };
    while(true) {
        badAccess(ar);
        Thread.sleep(2000);

        for (int i = 0; i < 5_000; i++) {
            byte[] unusedData = new byte[1024];
            for (int j = 0; j < 1024; j++) {
                // setting 9 here, so that it is easily recognized
                // if it is read by badAccess()
                unusedData[j] = 9;
            }
        }
    }
}

```

As can be seen, this code first loads the `badaccess` library (that is the native part of this bug seeding), declares the native `badAccess()` method, and then defines the `main()` method. Inside it, there is a byte array filled with ones, followed by an endless while loop that first calls the native `badAccess()` method with the byte array as an argument, then it sleeps for two seconds, and lastly, there is a for loop inside the while loop, which creates new byte arrays filled with nines. This is done to trigger the GC more often and to demonstrate that the buggy JNI code can access the nines even though it should not.

The C part of the bug seeding is in a file called `badaccess.c`, and it contains this:

```

#include <jni.h>
#include <stdio.h>

static jbyte* elems = NULL;
static jsize sz = 0;

/*
 * Class: de_stuefe_repros_BadAccess
 * Method: badAccess
 * Signature: ([B)V
 */
JNIEXPORT void JNICALL Java_de_stuefe_repros_BadAccess_badAccess
(JNIEnv* env, jclass clazz, jbyteArray ar)
{
    if (elems == NULL) {
        sz = (*env)->GetArrayLength(env, ar);
        jboolean is_copy = JNI_FALSE;

```

```

        elems = (*env)->GetPrimitiveArrayCritical(env, ar, &is_copy);
        (*env)->ReleasePrimitiveArrayCritical(env, ar, elems, is_copy);
    }

    printf("Accessing array on this address: %p\n", elems);
    for (int i = 0; i < sz; i++) {
        printf("%02x ", elems[i]);
        if (((i + 1) % 8) == 0) {
            printf("\n");
        }
    }
}

```

The first things after the includes in this file are the definitions of two static variables. The first one is a pointer to a Java byte array called `elems`, and the second one is the size of this array called `sz`. After that, the JNI method `badAccess()` is defined. The first time this method is called (`elems` is `NULL`), the array length is saved using the `GetArrayLength()` call, and a pointer to the array is saved using the `GetPrimitiveArrayCritical()` call. The `is_copy` argument is set to false, so that the pointer has direct access to the array. If it were set to true, it would only create a copy of the array to work with. As can be seen, the array is “released” right after that using the `ReleasePrimitiveArrayCritical()` call, meaning that any access to it afterward is not safe (and that is what this bug seeding is trying to exploit).

Apart from that, the JNI method also includes a simple for loop in which the contents of the array are accessed through the `elems` pointer, and the bytes it contains are printed. This loop is by purpose after the critical section, so accessing the array here is unsafe.

Now, these two files need to be compiled. The Java file can be compiled normally with `javac`, and the C files need to be compiled into a dynamic library, while including JNI header files, which are attached to the compiled JVM.

At this point, everything is ready to run (the path to the directory containing the compiled dynamic library needs to be provided as an argument after `-Djava.library.path=`, for it to run correctly with the native code). Without `SanitizeGC`, it runs without crashing. At first, it prints these statements, accessing the array with its expected contents (all ones):

```

Accessing array on this address: 0xffa69990
01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01

```

However, after less than nine seconds, a GC happens that collects the region containing the array. After that, it either prints unknown data or the nines, which are being allocated to fill the space:

```

Accessing array on this address: 0xffa69990
09 09 09 09 09 09 09 09
09 09 09 09 09 09 09 09

```

However, this is incorrect behavior, it should never be able to print anything other than the ones. The program was stopped manually after 90 seconds.

With `SanitizeGC` turned on, it also prints the array filled with ones a few times. And, same as without `SanitizeGC`, about nine seconds in, a GC occurs that collects the region containing the array. However, this time it does not continue executing. It crashes instead,

unable to print the array's contents after the collection, which should be the correct behavior. It also generates a crash log again. This crash log, apart from everything else, contains the SanitizeGC log line mentioned in the previous section, as well as the stack trace leading to the JNI code:

```
C [libbadaccess.so+0x11eb] Java_de_stuefe_repros_BadAccess_badAccess+0xd2
j de.stuefe.repros.BadAccess.badAccess([B)V+0
J 460% c2 de.stuefe.repros.BadAccess.main([Ljava/lang/String;)...
v ~StubRoutines::call_stub 0x00007f39f445fcfa
V [libjvm.so+0xa53ee6] JavaCalls::call_helper(...)+0x634
  (javaCalls.cpp:415)
```

From this second bug seeding, it can be seen that SanitizeGC might not only be useful on the HotSpot side, but it can also serve as a tool accessible to the end users who wish to verify that their code does not contain any bugs that may lead to memory vulnerabilities. While this exact bug example may seem simple to spot, in more complex codebases, the bug may be hidden more effectively and harder to spot and debug without a tool like SanitizeGC.

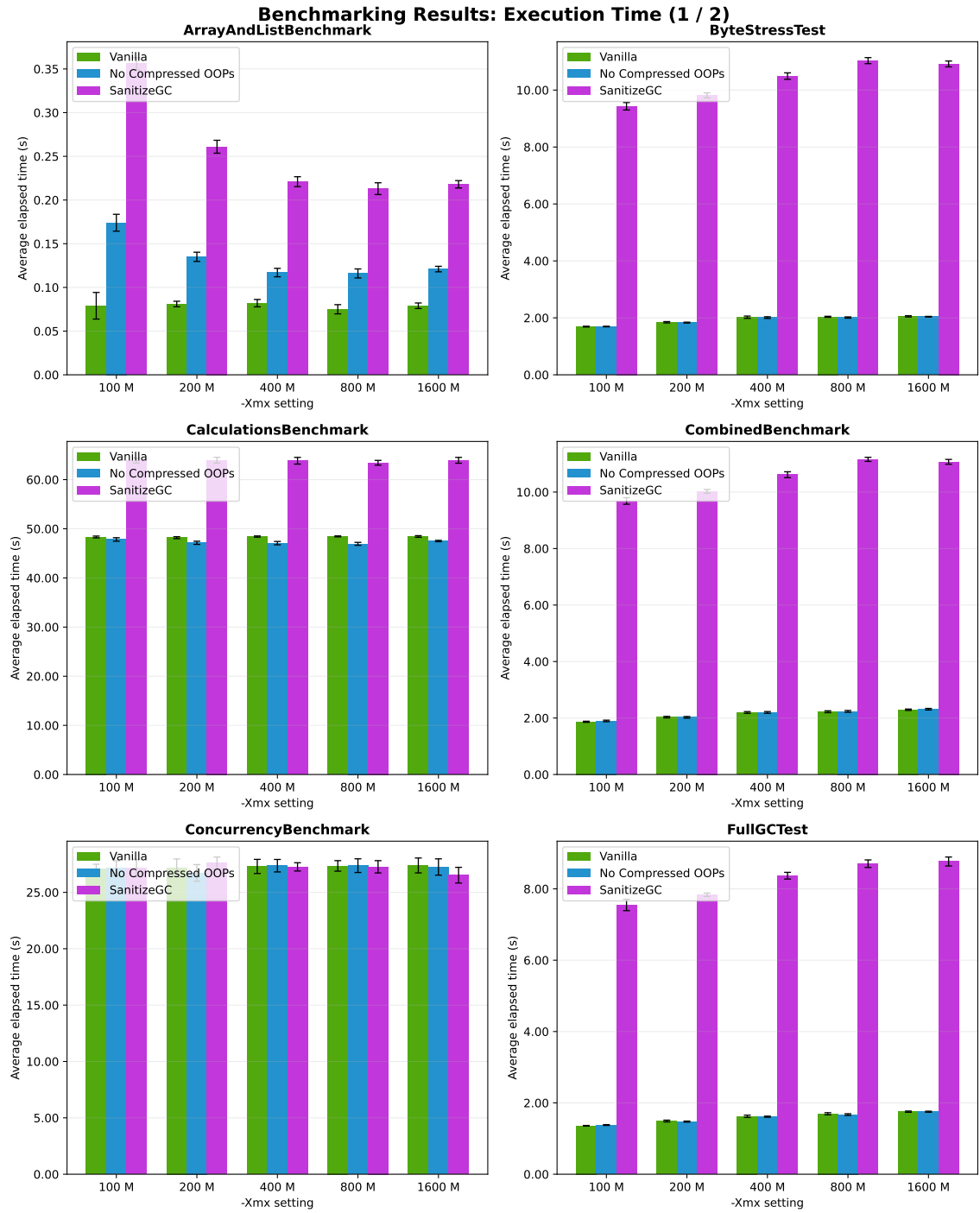


Figure 6.1: Execution time results of benchmarks (first part).

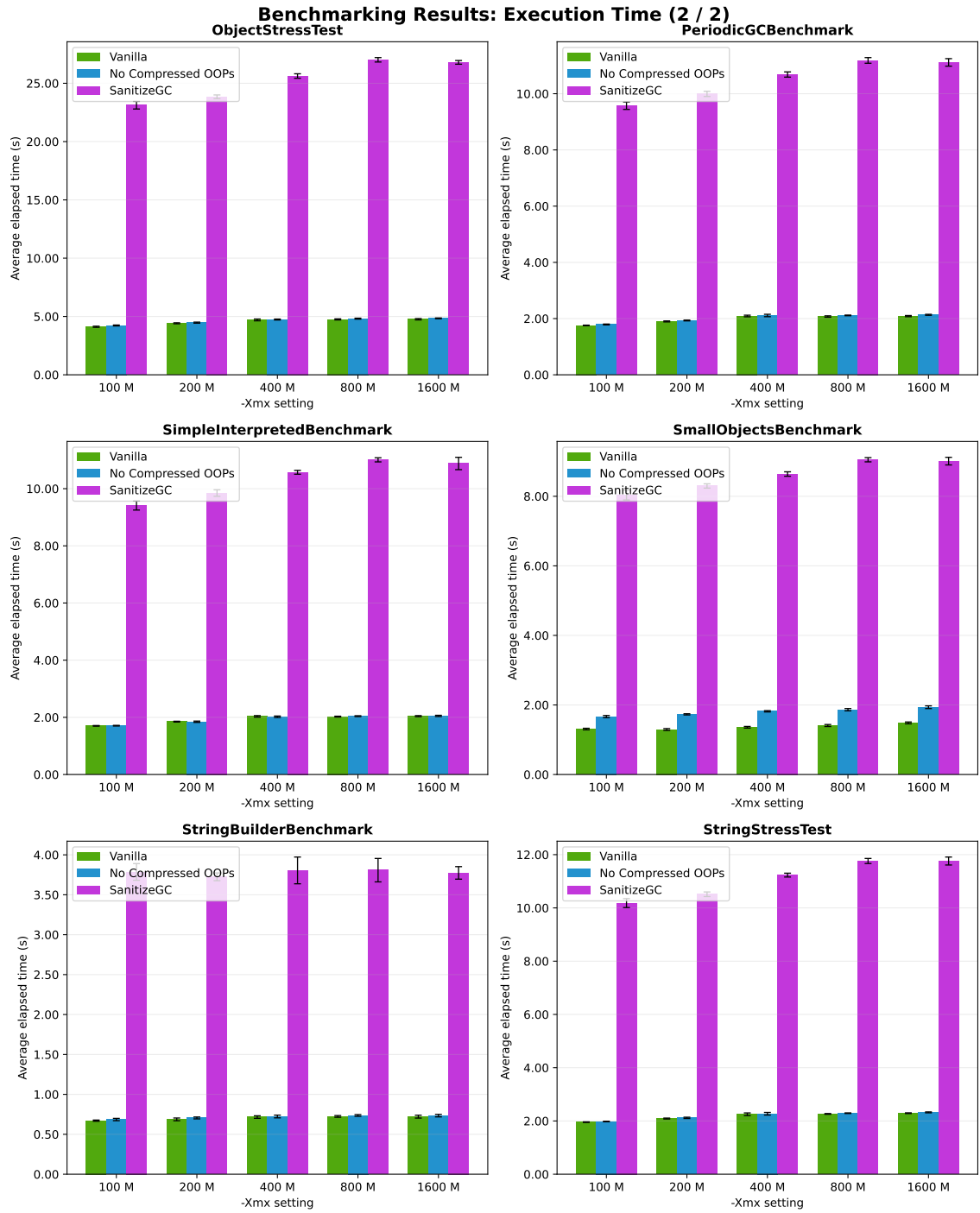


Figure 6.2: Execution time results of benchmarks (second part).

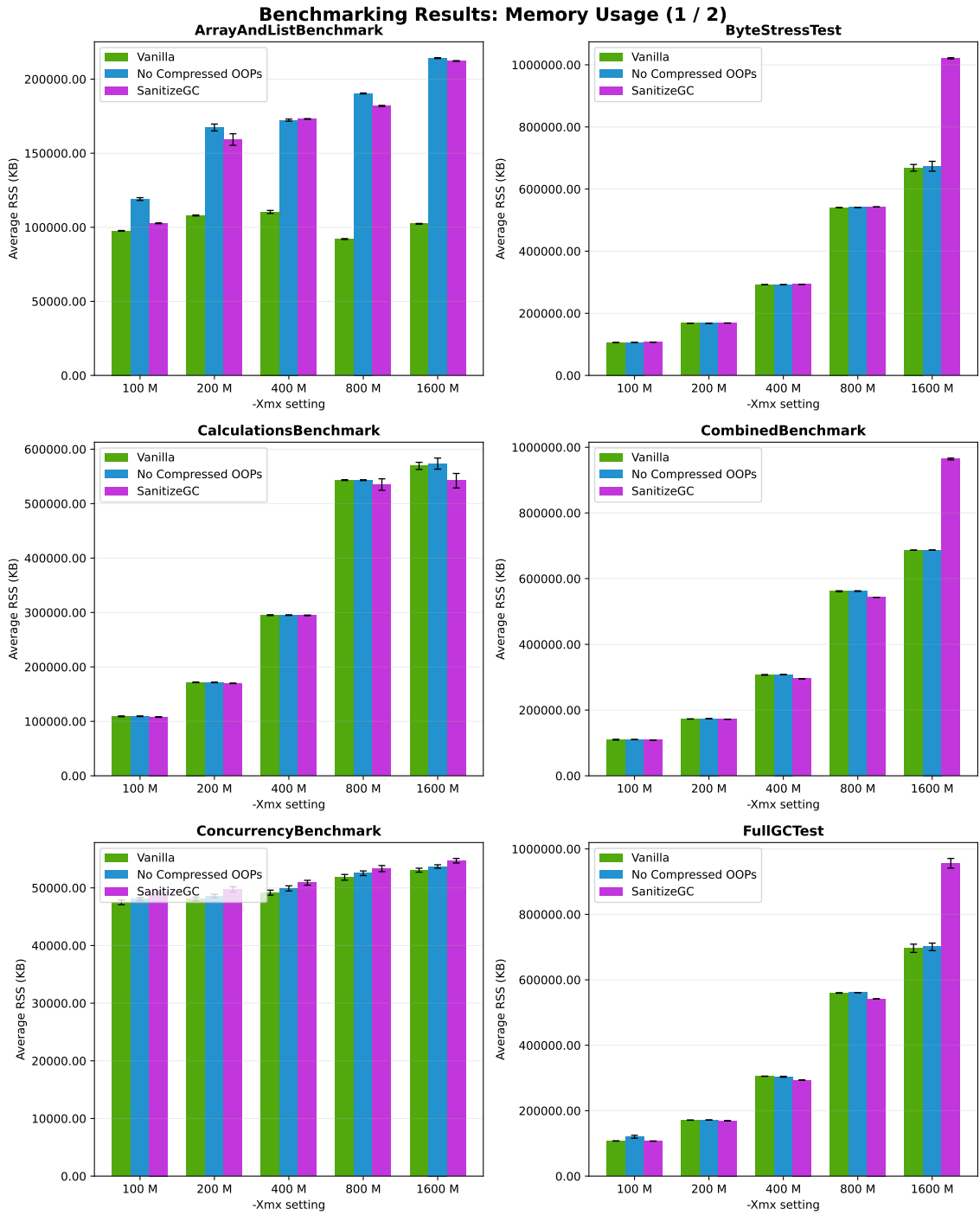


Figure 6.3: Memory usage results of benchmarks (first part).

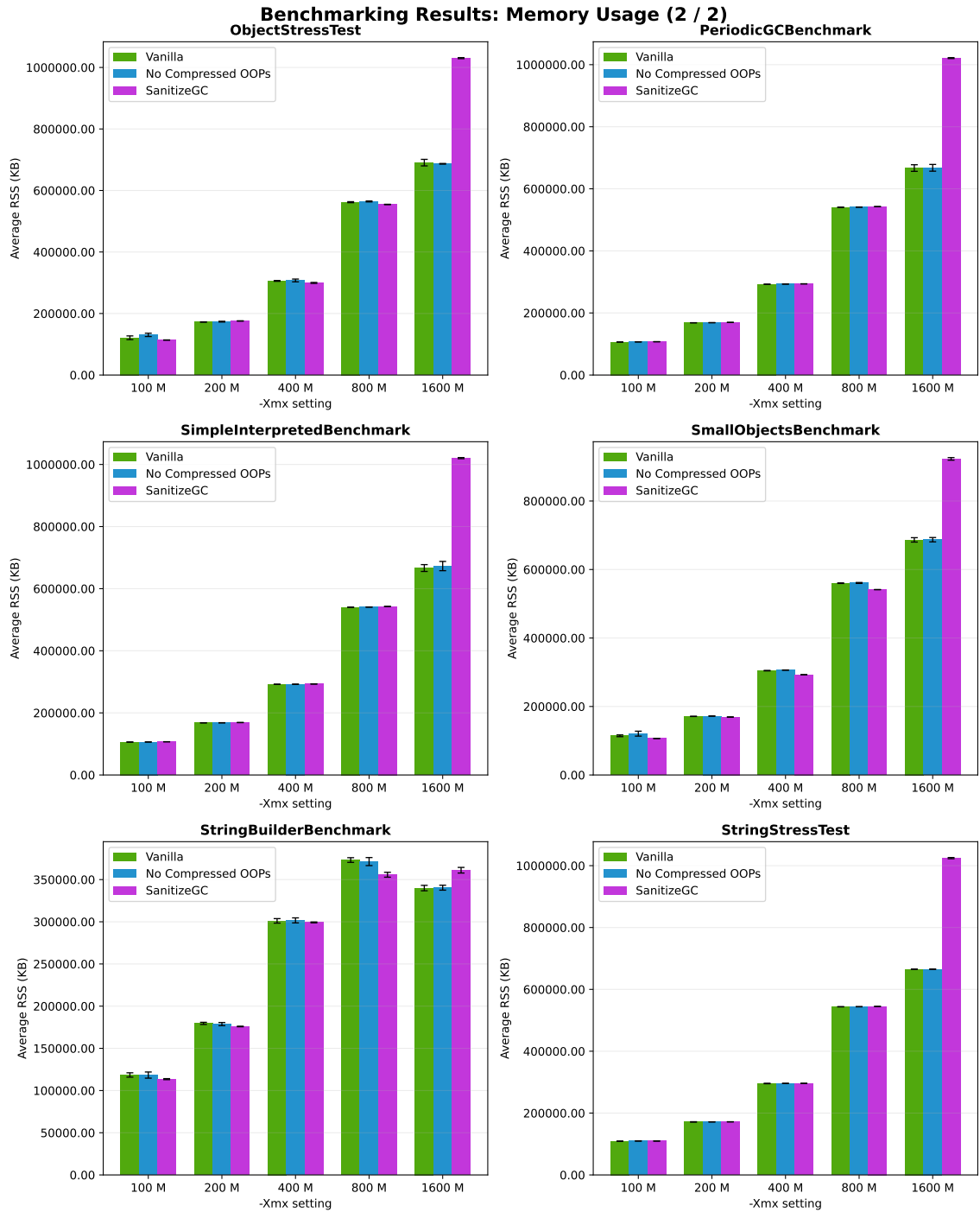


Figure 6.4: Memory usage results of benchmarks (second part).

Chapter 7

Conclusion

To sum up, this thesis presents the SanitizeGC extension of the Garbage-First garbage collector, which was designed and implemented. The extension was designed as a proof-of-concept diagnostic tool to detect memory vulnerabilities, especially use-after-free vulnerabilities caused by dangling pointers. This detection is done by moving the heap regions throughout memory and “locking” (by removing permissions) the old locations they occupied. This way, if a dangling-pointer dereference occurs, it is caught, and a stack trace is printed.

After implementation, the extension was tested against two test suites. The first one was created specifically for this project, and the second was the general G1 test suite already present in the codebase. Then, the extension was tested on a few real-world applications to further prove its functionality. Also, the performance of SanitizeGC was measured on high memory workload tests. The results showed that SanitizeGC was, on average, 4.5 times slower and used 1.1 times more memory than a JVM with SanitizeGC disabled.

The most important part of the testing, however, was the bug seeding that was done. There were two of them, and both ended in success. The first bug seeding introduces a bug into the JVM itself. While without SanitizeGC, the issue was not detected, with SanitizeGC, the issue was found immediately, even with a stack trace leading to it. The second bug seeding consisted of buggy JNI code. Again, without SanitizeGC, it was not detected, and with SanitizeGC, it was.

As for the next steps in this project, the first one will be to debug and fix the few failures that were found using the G1 test suite. As these failures seemed complex, this will most likely take some time. After the known failures are fixed, the next step would be to implement a solution for the humongous regions, as they have not been moved for now. This will require implementing a separate logic and address mapper for the humongous regions, as they are allocated differently from the normal regions.

After all of that is fixed/implemented, it would make sense to rebase the project on a newer Java version. At the time of writing this thesis, the newest version of Java is 26. However, an even newer version might be released in the meantime. The rebase itself will hopefully not be too difficult, as most of the changes are located in separate files.

And finally, after all of that is done, and if everything goes as planned, an effort will be made to share this project with the wider OpenJDK community, with the goal of possibly creating an official JDK Enhancement Proposal that results in the changes being eventually merged into HotSpot’s codebase.

In my personal opinion, the thesis as a whole ended in success. The main goal was to create a proof of concept of changes to showcase that something like this might be beneficial

to the G1 GC. And even with some limitations for now (the project being Linux-only, not supporting humongous regions, etc.), this project already shows that it has the potential to help with overall security, mainly during the development and testing phases of the JVM itself as a diagnostics tool (there might, for example, be a special test suite meant to be run during the overall testing with SanitizeGC enabled in it). However, it also showed potential in catching bugs in user-made JNI code. This thesis might serve as a reference in any future work on this.

Bibliography

- [1] BECKWITH, M. *Garbage First Garbage Collector Tuning* online. August 2013. Available at: <https://www.oracle.com/technical-resources/articles/java/g1gc.html>. [cit. 2025-06-04].
- [2] CABALLERO, J.; GRIECO, G.; MARRON, M. and NAPPA, A. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, July 2012, p. 133–143. ISSTA 2012. ISBN 9781450314541. Available at: <https://doi.org/10.1145/2338965.2336769>.
- [3] CISA. *The Urgent Need for Memory Safety in Software Products* online. 06. december 2023. Available at: <https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products>. [cit. 2025-06-11].
- [4] DETLEFS, D.; FLOOD, C.; HELLER, S. and PRINTEZIS, T. Garbage-first garbage collection. In: *Proceedings of the 4th International Symposium on Memory Management*. New York, NY, USA: Association for Computing Machinery, October 2004, p. 37–48. ISMM '04. ISBN 1581139454. Available at: <https://doi.org/10.1145/1029873.1029879>.
- [5] FLOOD, C. H.; KENNKE, R.; DINN, A.; HALEY, A. and WESTRELIN, R. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. New York, NY, USA: Association for Computing Machinery, August 2016. PPPJ '16. ISBN 9781450341356. Available at: <https://doi.org/10.1145/2972206.2972210>.
- [6] FREE SOFTWARE FOUNDATION. *GNU GENERAL PUBLIC LICENSE: Version 2* online. June 1991. Available at: <https://www.gnu.org/licenses/old-licenses/gpl-2.0-standalone.html>. [cit. 2026-05-15]. Standalone text of the license.
- [7] GITHUB STAFF. *Octoverse: A new developer joins GitHub every second as AI leads TypeScript to #1* online. 28. october 2025. Available at: <https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/>. [cit. 2026-01-16].
- [8] IBM CORPORATION. *What is Garbage Collection in Java?* online. Available at: <https://www.ibm.com/think/topics/garbage-collection-java>. [cit. 2026-01-16].

- [9] IBM CORPORATION. *Java virtual machine* online. 14. february 2025. Available at: <https://www.ibm.com/docs/en/sdk-java-technology/8?topic=introduction-java-virtual-machine>. [cit. 2026-01-16].
- [10] JANAPATI, S. P. R. *Understanding the Java Memory Model and Garbage Collection* online. 18. august 2016. Available at: <https://dzone.com/articles/understanding-the-java-memory-model-and-the-garbag>. [cit. 2025-06-04].
- [11] JONES, R.; HOSKING, A. and MOSS, E. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 2nd ed. New York, NY, USA: Chapman and Hall/CRC, june 2023. ISBN 9781003276142. Available at: <https://doi.org/10.1201/9781003276142>.
- [12] KORANDO, B. *The G1 Garbage Collector – Sip of Java* online. 20. june 2022. Available at: <https://inside.java/2022/06/20/sip056/>. [cit. 2026-01-16].
- [13] LINUX MANUAL PAGE AUTHORS. *System Calls Manual – mmap(2)* online. 6.16. October 2025. Available at: <https://www.man7.org/linux/man-pages/man2/mmap.2.html>. [cit. 2026-05-17].
- [14] LINUX MANUAL PAGE AUTHORS. *System Calls Manual – mprotect(2)* online. 6.16. October 2025. Available at: <https://www.man7.org/linux/man-pages/man2/mprotect.2.html>. [cit. 2026-05-17].
- [15] MICROSOFT CORPORATION. *Fundamentals of garbage collection* online. Available at: <https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals>. [cit. 2026-01-16].
- [16] MITRE CORPORATION. *CWE-415: Double Free* online. 11. december 2025. Available at: <https://cwe.mitre.org/data/definitions/415.html>. [cit. 2026-01-17].
- [17] MITRE CORPORATION. *CWE-416: Use After Free* online. 11. december 2025. Available at: <https://cwe.mitre.org/data/definitions/416.html>. [cit. 2026-01-17].
- [18] OPENJDK CONTRIBUTORS. *Building the JDK* online. 04. february 2026. Available at: <https://github.com/openjdk/jdk/commits/master/doc/building.md>. [cit. 2026-04-29].
- [19] OPENJDK WEBSITE CONTRIBUTORS. *The HotSpot Group* online. Available at: <https://openjdk.org/groups/hotspot/>. [cit. 2026-01-16].
- [20] OPENJDK WEBSITE CONTRIBUTORS. *OpenJDK* online. Available at: <https://openjdk.org/>. [cit. 2026-01-16].
- [21] OPENJDK WEBSITE CONTRIBUTORS. *Regression Test Harness for the JDK: jtreg* online. Available at: <https://openjdk.org/jtreg/>. [cit. 2026-04-29].
- [22] OPENJDK WIKI CONTRIBUTORS. *Main – ZGC – OpenJDK Wiki* online. 19. november 2025. Available at: <https://wiki.openjdk.org/spaces/zgc/overview>. [cit. 2026-05-15].

- [23] OPENSSEF. *Announcing the Release of “The Memory Safety Continuum”* online. 28. april 2025. Available at: <https://openssf.org/blog/2025/04/28/announcing-the-release-of-the-memory-safety-continuum/>. [cit. 2025-06-11].
- [24] ORACLE CORPORATION. *Garbage-First (G1) Garbage Collector* online. Available at: <https://docs.oracle.com/en/java/javase/25/gctuning/garbage-first-g1-garbage-collector1.html>. [cit. 2026-03-30].
- [25] ORACLE CORPORATION. *Garbage-First Garbage Collector* online. Available at: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc.html. [cit. 2026-01-19].
- [26] ORACLE CORPORATION. *Java Garbage Collection Basics* online. Available at: <https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>. [cit. 2025-06-09].
- [27] ORACLE CORPORATION. *Java Native Interface Specification: 1 – Introduction* online. Available at: <https://docs.oracle.com/en/java/javase/11/docs/specs/jni/intro.html>. [cit. 2026-05-03].
- [28] ORACLE CORPORATION. *The JIT Compile Process* online. 2010. Available at: <https://docs.oracle.com/cd/E19620-01/805-4031/ch1intro-10/index.html>. [cit. 2026-01-16].
- [29] OWASP FOUNDATION. *Buffer Overflow* online. Available at: https://owasp.org/www-community/vulnerabilities/Buffer_Overflow. [cit. 2026-01-18].
- [30] PIZLO, F.; PETRANK, E. and STEENSGAARD, B. A study of concurrent real-time garbage collectors. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery, june 2008, vol. 43, no. 6, p. 33–44. ISSN 0362-1340. Available at: <https://doi.org/10.1145/1379022.1375587>.
- [31] QUARKUS WEBSITE CONTRIBUTORS. *Quarkus – Supersonic Subatomic Java* online. Available at: <https://quarkus.io/>. [cit. 2026-05-02].
- [32] RR WEBSITE CONTRIBUTORS. *Rr: lightweight recording & deterministic debugging* online. Available at: <https://rr-project.org/>. [cit. 2026-04-16].
- [33] SEREBRYANY, K.; BRUENING, D.; POTAPENKO, A. and VYUKOV, D. AddressSanitizer: a fast address sanity checker. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USA: USENIX Association, June 2012, p. 28–38. USENIX ATC’12. Available at: <https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf>.
- [34] SHIPILĚV, A. *JVM Anatomy Quark #4: TLAB allocation* online. 03. march 2019. Available at: <https://shipilev.net/jvm/anatomy-quarks/4-tlab-allocation/>. [cit. 2026-05-16].
- [35] THE KERNEL DEVELOPMENT COMMUNITY. *Concepts overview* online. Available at: <https://docs.kernel.org/admin-guide/mm/concepts.html>. [cit. 2026-04-01].

- [36] TIOBE. *TIOBE Index for January 2026* online. January 2026. Available at: <https://www.tiobe.com/tiobe-index/>. [cit. 2026-01-27].
- [37] WIKIPEDIA CONTRIBUTORS. *Java (programming language)* online. 14. january 2026. Available at: [https://en.wikipedia.org/wiki/Java_\(programming_language\)](https://en.wikipedia.org/wiki/Java_(programming_language)). [cit. 2026-01-16].
- [38] WIKIPEDIA CONTRIBUTORS. *X86-64* online. 16. march 2026. Available at: <https://en.wikipedia.org/wiki/X86-64>. [cit. 2026-04-01].

Appendix A

Attachment Directory Structure

The attachment of this thesis contains these directories:

- **benchmarks** – This directory contains Python scripts used for running the benchmarks and generating the resulting graphs. It also contains three CSV files with benchmarking results, four PDFs with graphs, and an ODS file with calculations on the results.
- **binaries** – This directory contains two different directories with **release** and **slow debug** builds of the SanitizeGC JDK. The actual `java/javac/jar/...` binaries are located in `binaries/linux-x86_64-{release,slowdebug}/bin`.
- **bugseeding** – Inside of this directory, there are two more, **hotspot** and **jni**. Inside each of those are the files needed to recreate the bugseeding done in chapter 6. Both of those also contain a **logs** directory with outputs from the bugseeding and a crash log when SanitizeGC was enabled.
- **doc_src** – Contains the LaTeX source code for the written part of the thesis, as well as all the figures that were used.
- **src** – This contains the source code of the extended HotSpot’s codebase. It is basically a code cloned from the `rh-openjdk/SanitizerGC` repository, branch `sanitizerc-jdk23`¹, without the `.git` directory to save space. It also contains a file called `sanitizerc.diff`, which contains an output of the `git diff` command against the non-touched `master` branch.
- **test_logs** – Contains two directories, **release** and **slowdebug**. Both of those contain the output of `jtrg` testing on the SanitizeGC and G1 test suites (the outputs consist of reports and individual test outputs and results).
- **tools** – Contains a pre-built “boot JDK” needed for building (downloaded from Adoptium²) in the `boot_jdk` directory. It also contains the pre-built `jtrg` tool used for testing (downloaded from Aleksey Shipilëv’s website³) in the `jtrg` directory.

¹<https://github.com/rh-openjdk/SanitizeGC/tree/sanitizerc-jdk23>

²<https://adoptium.net/temurin/releases?version=23&mode=filter&os=linux&arch=x64>

³<https://builds.shipilev.net/jtrg/>

Appendix B

Building and Running the SanitizeGC

In this chapter, the building and running of the SanitizeGC is described. All of these steps were tested on a clear install of Debian 13, but the building should also work similarly on other Linux distributions. Also, all of these steps assume the code from the attachment is used, the code in the repository might potentially change in the meantime.

B.1 Building SanitizeGC

To build the JDK, you first need to install dependencies:

```
sudo apt install build-essential autoconf zip libasound2-dev libcups2-dev \
    libfontconfig1-dev libx11-dev libxext-dev libxrender-dev libxrandr-dev \
    libxtst-dev libxt-dev
```

The next step is setting the `JAVA_HOME` system variable to the directory with the boot JDK:

```
export JAVA_HOME=<PATH_TO_THE_ATTACHMENT>/tools/boot_jdk
```

Now, you can create a configuration for the build (assuming you are located in the `src` directory of the attachment):

```
bash configure --with-debug-level=slowdebug
```

This will create a configuration for the `slowdebug` build. You can replace `slowdebug` with `release` if you wish to (or you can even run this command twice with two different build types).

Now, to initiate the build, you can run this:

```
make exploded-image CONF=slowdebug
```

If you chose the `release` debug level in the previous step, put it there instead (or run this twice, if you choose both configurations). The `exploded-image` builds the minimal image required to run the JDK, so it should be the fastest. If you wish to export the built JDK, you might replace it with the `images` target instead (however, the HotSpot bug seeding below requires the `exploded-image` build).

After it finishes, the built JDK will be located inside the `build/linux-x86_64-server-{release,slowdebug}/jdk` directory.

B.2 Running Test Suites

To run the test suites, first move into an empty directory (it does not have to be empty, but directories with reports will be created at the current location). Now, you can run the SanitizeGC test suite using the jtreg utility like this:

```
<PATH_TO_THE_ATTACHMENT>/tools/jtreg/bin/jtreg -v:default \  
-jdk:<PATH_TO_THE_ATTACHMENT>/src/build/linux-x86_64-server-  
{release,slowdebug}/jdk/ \  
<PATH_TO_THE_ATTACHMENT>/src/test/hotspot/jtreg/gc/SanitizeGC
```

This will create two directories: `JTreport` and `JTWork`. Inside `JTreport`, there is a `index.html` file with the report of the results. Inside `JTWork`, there are the logs of individual tests.

To run the G1 test suite, the command is a bit different, as the SanitizeGC needs to be manually set:

```
<PATH_TO_THE_ATTACHMENT>/tools/jtreg/bin/jtreg -v:default \  
-jdk:<PATH_TO_THE_ATTACHMENT>/src/build/linux-x86_64-server-  
{release,slowdebug}/jdk/ \  
-javaoption:-XX:+SanitizeGC \  
<PATH_TO_THE_ATTACHMENT>/src/test/hotspot/jtreg/gc/g1
```

This will also create the two directories, if they are not already present.

B.3 Running Benchmarks

First, install the dependencies of the benchmarks:

```
sudo apt install python3-pip time  
pip install numpy pandas matplotlib --break-system-packages
```

Now, navigate into the `src` directory and compile the benchmarks (and other tests) using this command:

```
test/hotspot/jtreg/gc/SanitizeGC/compile.sh
```

Next, navigate into the `benchmarks` directory. You can remove the old results there like this:

```
rm *.csv *.pdf
```

Now, you need to edit the `run_benchmarks.py` file on lines 6 and 9. On line 6, you need to set the path to the compiled SanitizeGC `jdk` directory (running benchmarks only makes sense on the `release` build, so set it there), and on line 9, you need to set the path to the SanitizeGC source code (the `src` directory of the attachment).

To start the benchmarks, run (it will probably take a lot of time):

```
python3 run_benchmarks.py
```

After it finishes, you can plot the results using this:

```
python3 plot_benchmarks.py
```

B.4 Running Bug Seeding

Below is a description of how to run both bug seedings that were described in chapter 6. Both of these assume the `slowdebug` build is used (as it is better for debugging), but it should also work with `release` builds after tweaking some values.

B.4.1 HotSpot Bug Seeding

For the HotSpot bug seeding, first make sure you have already compiled the `slowdebug` build with the `exploded-image` target (this bug seeding forces garbage collection by allocating garbage bytes, and that might cause a problem if the whole compilation is done, as the bug seeded code is run during it). Now, if you are located in the `src` directory of the attachment, get the Java test program ready first:

```
build/linux-x86_64-server-slowdebug/jdk/bin/javac \  
  ../bugseeding/hotspot/Program.java
```

Next, you can run this command to apply the buggy patch:

```
patch src/hotspot/share/classfile/javaClasses.cpp \  
  ../bugseeding/hotspot/introduce_bug.patch
```

Then recompile again (without cleaning):

```
make exploded-image CONF=slowdebug
```

And now try executing the program with the buggy java binary without SanitizeGC:

```
build/linux-x86_64-server-slowdebug/jdk/bin/java \  
  -cp ../bugseeding/hotspot/ Program
```

If you get an error saying that a GC triggered before the VM initialization completed during the java execution, you probably need to edit the `bugseeding/hotspot/introduce_bug.patch` file to lower the amount of garbage bytes being allocated. You can do this by first reverting the patch:

```
patch -R src/hotspot/share/classfile/javaClasses.cpp \  
  ../bugseeding/hotspot/introduce_bug.patch
```

And then by actually lowering the value by changing the number of iterations of the for loop (fourth line in the patch file) from 250 to, for example, 100. The 250 value was chosen on my machine, but it will require some experimentation on different machines, as the threshold for triggering GC is nondeterministic. After you change the value, you can continue from the patch step again.

Once you have that sorted out, you can try running the java binary again and checking the output for lines where the addresses do not match. If you locate one, you have proof that a value was written to the wrong memory. The next step is to run it with SanitizeGC enabled:

```
build/linux-x86_64-server-slowdebug/jdk/bin/java -XX:+SanitizeGC \  
  -cp ../bugseeding/hotspot/ Program
```

If a wrong write is done, it will crash and generate a crash log. It will also print a simple message that the problematic frame is `oopDesc::int_field_put()`.

B.4.2 JNI Bug Seeding

If you have done the HotSpot bug seeding, discard the changes first (assuming you are in the `src` directory again):

```
patch -R src/hotspot/share/classfile/javaClasses.cpp \  
  ../bugseeding/hotspot/introduce_bug.patch
```

And recompile:

```
make exploded-image CONF=slowdebug
```

Now, navigate into the `bugseeding/jni` directory. Inside this directory, there are two files that need editing, the `build.sh` and `run.sh`. In both of those, you need to add the path to the compiled JDK on line 3. The path looks like this:

```
<PATH_TO_THE_ATTACHMENT>/src/build/linux-x86_64-server-slowdebug/jdk
```

Then, you need to make the files executable:

```
chmod +x build.sh run.sh
```

And finally, you can run the `build.sh` script that builds everything necessary, and the `run.sh` script that runs the bug seeding with SanitizeGC enabled. After a few seconds, the program should crash, generating another crash log containing a stack trace pointing to the JNI code. If you wish, you can remove the `-XX:+SanitizeGC` switch from the `run.sh` file to run it without SanitizeGC enabled. It will not crash, and after a while, you will see that the JNI method reads data it should not be able to read.