Use at Your Own Risk: The Java Unsafe API in the Wild

Luis Mastrangelo Luca Ponzanelli Andrea Mocci
Michele Lanza Matthias Hauswirth Nathaniel Nystrom
Faculty of Informatics, Università della Svizzera italiana (USI), Switzerland
{first.last}@usi.ch

Abstract

Java is a safe language. Its runtime environment provides strong safety guarantees that any Java application can rely on. Or so we think. We show that the runtime actually does not provide these guarantees—for a large fraction of today's Java code. Unbeknownst to many application developers, the Java runtime includes a "backdoor" that allows expert library and framework developers to circumvent Java's safety guarantees. This backdoor is there by design, and is well known to experts, as it enables them to write high-performance "systems-level" code in Java.

For much the same reasons that safe languages are preferred over unsafe languages, these powerful—but unsafe—capabilities in Java should be restricted. They should be made safe by changing the language, the runtime system, or the libraries. At the very least, their use should be restricted. This paper is a step in that direction.

We analyzed 74 GB of compiled Java code, spread over 86,479 Java archives, to determine how Java's unsafe capabilities are used in real-world libraries and applications. We found that 25% of Java bytecode archives depend on unsafe third-party Java code, and thus Java's safety guarantees cannot be trusted. We identify 14 different usage patterns of Java's unsafe capabilities, and we provide supporting evidence for why real-world code needs these capabilities. Our long-term goal is to provide a foundation for the design of new language features to regain safety in Java.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Patterns

General Terms Design, Languages, Measurement

Keywords unsafe, patterns, mining, Java, Maven Central, Stack Overflow

1. Introduction

The Java Virtual Machine (JVM) executes Java bytecode and provides other services for programs written in many programming languages, including Java, Scala, and Clojure. The JVM was designed to provide strong safety guarantees. However, many widely used JVM implementations expose an API that allows the developer to access low-level, unsafe features of the JVM and underlying hardware, features that are unavailable in safe Java bytecode. This API is provided through an undocumented[1] class, *sun.misc.Unsafe*, in the Java reference implementation produced by Oracle.

Other virtual machines provide similar functionality. For example, the C# language provides an `unsafe` construct on the .NET platform[2], and Racket provides unsafe operations[3].

The operations *sun.misc.Unsafe* provides can be dangerous, as they allow developers to circumvent the safety guarantees provided by the Java language and the JVM. If misused, the consequences can be resource leaks, deadlocks, data corruption, and even JVM crashes.[4] [5] [6] [7] [8]

We believe that *sun.misc.Unsafe* was introduced to provide better performance and more capabilities to the writers of the Java runtime library. However, *sun.misc.Unsafe* is increasingly being used in third-party frameworks and libraries. Application developers who rely on Java's safety guarantees have to trust the implementers of the language runtime environment (including the core runtime libraries). Thus the use of *sun.misc.Unsafe* in the runtime libraries is no more risky than the use of an unsafe language to implement the JVM. However, the fact that more and more "normal" libraries are using *sun.misc.Unsafe* means that application developers have to trust a growing community of third-party

[1] `http://www.oracle.com/technetwork/java/faq-sun-packages-142232.html`

[2] `https://msdn.microsoft.com/en-us/en-en/library/chfa2zb8(v=vs.90).aspx`

[3] `http://docs.racket-lang.org/reference/unsafe.html`

[4] `https://groups.google.com/d/msg/elasticsearch/Nh-kXI5J6Ek/WXIZKhhGVHkJ`

[5] `https://github.com/EsotericSoftware/kryo/issues/219`

[6] `https://github.com/dain/snappy/issues/24`

[7] `https://netbeans.org/bugzilla/show_bug.cgi?id=229655`

[8] `https://netbeans.org/bugzilla/show_bug.cgi?id=244914`

Java library developers to not inadvertently tamper with the fragile internal state of the JVM.

Given that the benefits of safe languages are well known, and the risks of unsafe languages are obvious, why exactly does one need unsafe features in third-party libraries? Are those features used in real-world code? If yes, how are they used, and what are they used for?

We studied a large repository of Java code, Maven Central, to answer these questions. We analyzed 74 GB of compiled Java code, spread over 86,479 Java libraries, to determine the usage and impact of *sun.misc.Unsafe*. Our goal is to provide a strong foundation for informed decisions in the future evolution of the Java language and virtual machine.

The rest of this paper is organized as follows. Section 2 presents the concrete risks of using *sun.misc.Unsafe*. Section 3 discusses our research questions and introduces our study. Section 4 presents an overview of how *Unsafe* is used. Section 5 describes our analysis of the Stack Overflow question/answer database. Section 6 describes our methodology for finding *Unsafe* usage patterns. Sections 7 and 8 introduce and discuss the patterns we found. Section 9 presents related work, and Section 10 concludes the paper.

2. The Risks of Compromising Safety

We outline the risks of *Unsafe* by illustrating how the improper use of *Unsafe* violates Java's safety guarantees.

In Java, the unsafe capabilities are provided as instance methods of class *sun.misc.Unsafe*. Access to the class has been made less than straightforward. Class *sun.misc.Unsafe* is final, and its constructor is not public. Thus, creating an instance requires some tricks. For example, one can invoke the private constructor via reflection. This is not the only way to get hold of an unsafe object, but it is the most portable.

```
Constructor<Unsafe> c = Unsafe.class.
    getDeclaredConstructor();
c.setAccessible(true);
Unsafe unsafe = c.newInstance();
```

Listing 1. Instantiating an Unsafe object

Given the unsafe object, one can now simply invoke any of its methods to directly perform unsafe operations.

2.1 Violating Type Safety

In Java, variables are strongly typed. For example, it is impossible to store an int value inside a variable of a reference type. *Unsafe* can violate that guarantee: it can be used to store a value of any type in a field or array element.

```
class C {
  private Object f = new Object();
}
long fieldOffset = unsafe.objectFieldOffset(
        C.class.getDeclaredField("f") );
C o = new C();
unsafe.putInt(o, fieldOffset, 1234567890);
    // f now points to nirvana
```

Listing 2. *sun.misc.Unsafe* can violate type safety

2.2 Crashing the Virtual Machine

A quick way to crash the VM is to free memory that is in a protected address range, for example by calling `freeMemory` as follows.

```
unsafe.freeMemory(1);
```

Listing 3. *sun.misc.Unsafe* can crash the VM

In Java, the normal behavior of a method to deal with such situations is to throw an exception. Being unsafe, instead of throwing an exception, this invocation of `freeMemory` crashes the VM.

2.3 Violating Method Contracts

In Java, a method that does not declare an exception cannot throw any checked exceptions. *Unsafe* can violate that contract: it can be used to throw a checked exception that the surrounding method does not declare or catch.

```
void m() {
  unsafe.throwException(new Exception());
}
```

Listing 4. *sun.misc.Unsafe* can violate a method contract

2.4 Uninitialized Objects

Java guarantees that an object allocation also initializes the object by running its constructor. *Unsafe* can violate that guarantee: it can be used to allocate an object without ever running its constructor. This can lead to objects in states that the objects' classes would not seem to admit.

```
class C {
  private int f;
  public C() { f = 5; }
  public int getF() { return f; }
}

C c = (C)unsafe.allocateInstance(C.class);
assert c.getF()==5; // violated
```

Listing 5. *sun.misc.Unsafe* can lead to uninitialized objects

2.5 Monitor Deadlock

Java provides synchronized methods and synchronized blocks. These constructs guarantee that monitors entered at the beginning of a section of code are exited at the end. *Unsafe* can violate that contract: it can be used to asymmetrically enter or exit a monitor, and that asymmetry might be not immediately obvious.

```
void m() {
  unsafe.monitorEnter(o);
  if (c) return;
  unsafe.monitorExit(o);
}
```

Listing 6. *sun.misc.Unsafe* can lead to monitor deadlocks

The above examples are just the most straightforward violations of Java's safety guarantees. The *sun.misc.Unsafe*

class provides a multitude of methods that can be used to violate most guarantees Java provides.

To sum it up: *Unsafe* is dangerous. But should anybody care? In the next sections we present a study to determine whether and how *Unsafe* is used in real-world third-party Java libraries, and to what degree real-world applications directly and indirectly depend on it.

3. Overview of Our Study

We believe we should care about the dangers of *Unsafe* if the third-party usage of *Unsafe* could impact common application code. We want to answer the following questions:

*Q*1 : **Does *Unsafe* impact common application code?** We want to understand to what extent third-party code actually uses *Unsafe*.

*Q*2 : **Which features of *Unsafe* are used?** As *Unsafe* provides many features, we want to understand which ones are actually used, and which ones can be ignored.

*Q*3 : **Why are *Unsafe* features used?** We want to investigate what functionality third-party libraries require from *Unsafe*. This could point out ways in which the Java language and/or the JVM need to be evolved to provide the same functionality, but in a safer way.

*Q*4 : **What problems do developers who use *Unsafe* encounter?** If *Unsafe* is not just dangerous, but also confusing or difficult to use, then its use by third-party developers is particularly problematic. If there are specific *Unsafe* features or usage patterns that developers worry about, it would make sense to evolve Java or the JVM to provide safer alternatives in that direction.

To answer the above questions, we need to determine whether and how *Unsafe* is actually used in real-world third-party Java libraries, and to what degree real-world applications directly and indirectly depend on such unsafe libraries. To achieve our goal, several elements are needed.

Code Repository. As a code base representative of the "real world", we have chosen the Maven Central[9] software repository. The rationale behind this decision is that a large number of well-known Java projects deploy to Maven Central using Apache Maven[10]. Besides code written in Java, projects written in Scala are also deployed to Maven Central using the Scala Build Tool (sbt)[11]. Moreover, Maven Central is the largest Java repository[12], and it contains projects from the most popular source code management repositories, like GitHub[13] and SourceForge[14].

Artifacts. In Maven terminology, an artifact is the output of the build procedure of a project. An artifact can be any type of file, ranging from a `.pdf` to a `.zip` file. However, artifacts are usually `.jar` files, which archive compiled Java bytecode stored in `.class` files.

Bytecode Analysis. We examine these kinds of artifacts to analyze how they use *sun.misc.Unsafe*. We use a bytecode analysis library to search for method call sites and field accesses of the *sun.misc.Unsafe* class.

Dependency Analysis. We define the impact of an artifact as how many artifacts depend on it, either directly or indirectly. This helps us to define the impact of artifacts that use *sun.misc.Unsafe*, and thus the impact *sun.misc.Unsafe* has on real-world code overall.

Usage Pattern Detection. After all call sites and field accesses are found, we analyze this information to discover usage patterns. It is common that an artifact exhibits more than one pattern. Our list of patterns is not exhaustive. We have manually investigated the source code of the 100 highest-impact artifacts using *sun.misc.Unsafe* to understand why and how they are using it.

Stack Overflow Analysis. We studied problems encountered using *sun.misc.Unsafe* by analyzing the Stack Overflow question/answer database. After discovering usage patterns in the Maven archive, we use Stack Overflow to correlate them to discussions. Our goal is to understand the difficulties in implementing certain *Unsafe* usage patterns.

4. Is *Unsafe* Used?

In this section we answer our first two research questions: whether *Unsafe* impacts common application code, and which features of *Unsafe* are actually used.

We do this by mining Maven Central. The complete scripts and results used for this study are available online[15].

In the Maven Central repository it is possible to find language implementations, such as *org.scala-lang*:*scala-library*, *org.jruby*:*jruby-core*, *org.codehaus.groovy*:*groovy-all*, *org.python*:*jython*, and *com.oracle*:*truffle*. We treat them separately because they are not common application code, they are language implementations. The users of these languages trust these libraries as a Java user trust Java's implementation.

4.1 Gathering Artifacts

The complete Maven Central repository contains 959,300 artifacts, 106,574 unique artifacts—artifacts are versioned—and consists of ca. 1.7 TB of data[16]. For our analysis we only look at the last version of each artifact to search for *sun.misc.Unsafe* uses. Moreover we are only interested in a subset of this data: artifacts that archive compiled bytecode (`.class` files) e.g., `.jar`, `.war`, `.ear`, and `.ejb` files.

[9] `http://central.sonatype.org/`

[10] `http://maven.apache.org/`

[11] `http://www.scala-sbt.org/`

[12] `http://www.modulecounts.com/`

[13] `https://github.com/`

[14] `http://sourceforge.net/`

[15] `https://bitbucket.org/acuarica/java-unsafe-analysis`

[16] `http://search.maven.org/#stats`

We downloaded all artifacts subject to analysis from differents mirrors of Maven Central, including the ibiblio Digital Archive[17]. We downloaded the archive between May 29th and June 3rd, 2015. The downloaded repository consists of about 74 GB of data from 86,479 unique artifacts.

4.2 Determining Usage

To search for *sun.misc.Unsafe* static use, we mined bytecode using the following facts about the *sun.misc.Unsafe* class: *a*) it is declared as `final`; *b*) it inherits directly from `java.lang.Object`; *c*) its public methods (except for `getUnsafe`) are instance methods; and *d*) its public fields are declared as `static final`.

We implement our analysis on top of ASM [14]. Our analysis finds all virtual method invocation sites where the call target is of type *sun.misc.Unsafe*, and all static reads of fields of class *sun.misc.Unsafe*.

Sometimes *sun.misc.Unsafe* is used through reflection to avoid compilation dependencies (given that *sun.misc.Unsafe* is not part of the public API). Our study is restricted to static uses of *sun.misc.Unsafe*, which is a limitation of our work.

Our analysis found 48,490 uses of *sun.misc.Unsafe*—48,139 call sites and 351 field accesses—distributed over 817 different artifacts. This initial result shows that *Unsafe* is indeed used in third-party code.

4.3 Determining Impact

Unsafe does not only impact the artifacts that use it, but it transitively impacts all artifacts depending on those artifacts. We thus need to determine the transitive closure of unsafety.

Maven projects are described using POM files, which may contain dependency information. 47,127 of the artifacts we downloaded include such dependency information.

In Maven, dependencies have a scope. For example, a library artifact may depend on a testing framework artifact only for the purpose of testing. It will not depend on the testing framework for production runs.

We use the dependency information to determine the impact of the artifacts that use *sun.misc.Unsafe*. We rank all artifacts according to their impact (the number of artifacts that directly or indirectly depend on them). High-impact artifacts are important; a safety violation in them can affect any artifact that directly or indirectly depends on them. We find that while overall about 1% of artifacts directly use *Unsafe*, for the top-ranked 1000 artifacts, 3% directly use Unsafe. Thus, *Unsafe* usage is particularly prevalent in high-impact artifacts, artifacts that can affect many other artifacts.

Moreover, we found that 21,297 artifacts (47% of the 47,127 artifacts with dependency information, or 25% of the 86,479 artifacts we downloaded) directly or indirectly depend on *sun.misc.Unsafe*. Excluding language artifacts, numbers do not change much: Instead of 21,297 artifacts, we found 19,173 artifacts. 41% of the artifacts with dependency information, or 22% of artifacts downloaded. Thus, *sun.misc.Unsafe* usage in third-party code indeed impacts a large fraction of projects.

One limitation of our work is we only use the information stored in POM files to determine dependencies. Notice that this is an overapproximation because a POM file might be outdated, and the artifact itself could not use the dependency. Or it might happen that the dependency is used only on some program paths or under certain configurations. To precisely determine the artifact's dependencies, a more powerful and costly static or dymamic analysis would be needed.

4.4 Which Features of *Unsafe* Are Actually Used?

Figures 1 and 2 show all instance methods and static fields of *sun.misc.Unsafe*. For each member we show how many call sites or field accesses we found across the artifacts. The class provides 120 public instance methods and 20 public fields (version 1.8 update 40). The figure only shows 93 methods because the 18 methods in the *Heap Get* and *Heap Put* groups, and *staticFieldBase* are overloaded, and we combine overloaded methods into one bar.

We show two columns, *Application* and *Language*. The *Language* column corresponds to language implementation artifacts while the *Application* column corresponds to the rest of the artifacts.

We categorized the members into groups, based on the functionality they provide:

- The *Alloc* group contains only the *allocateInstance* method, which allows the developer to allocate a Java object without executing a constructor. This method is used 181 times: 180 in *Application* and 1 in *Language*.
- The *Array* group contains methods and fields for computing relative addresses of array elements. The fields were added as a simpler and potentially faster alternative in a more recent version of *Unsafe*. The value of all fields in this group are constants initialized with the result of a call to either *arrayBaseOffset* or *arrayIndexScale* in the *Array* group. The figures show that the majority of sites still invoke the methods instead of accessing the corresponding constant fields.
- The *CAS* group contains methods to atomically compare-and-swap a Java variable. These operations are implemented using processor-specific atomic instructions. For instance, on *x86* architectures, *compareAndSwapInt* is implemented using the `CMPXCHG` machine instruction. Figure 1 shows that these methods represent the most heavily used feature of *Unsafe*.
- Methods of the *Class* group are used to dynamically load and check Java classes. They are rarely used, with *defineClass* being used the most.
- The methods of the *Fence* group provide memory fences to ensure loads and stores are visible to other threads. These methods are implemented using processor-specific

[17] `http://mirrors.ibiblio.org/maven2/`

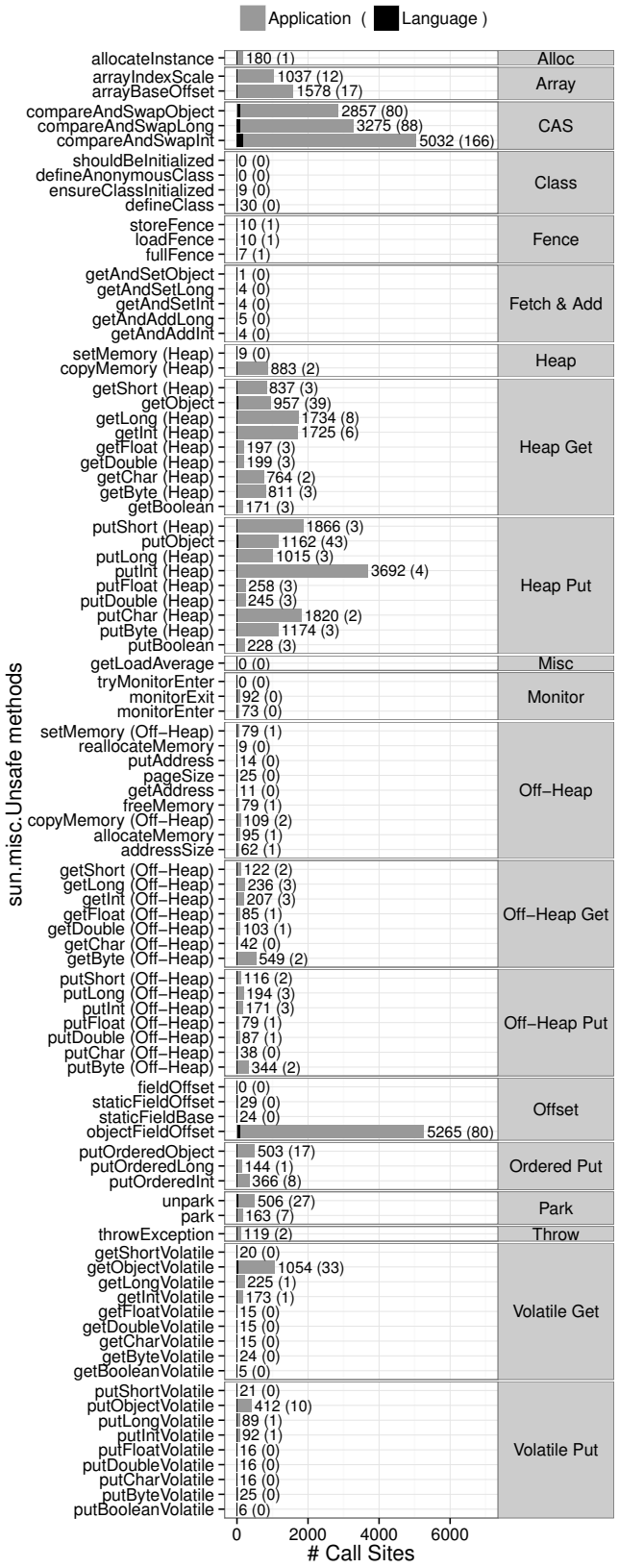

Figure 1. *sun.misc.Unsafe* method usage on Maven Central

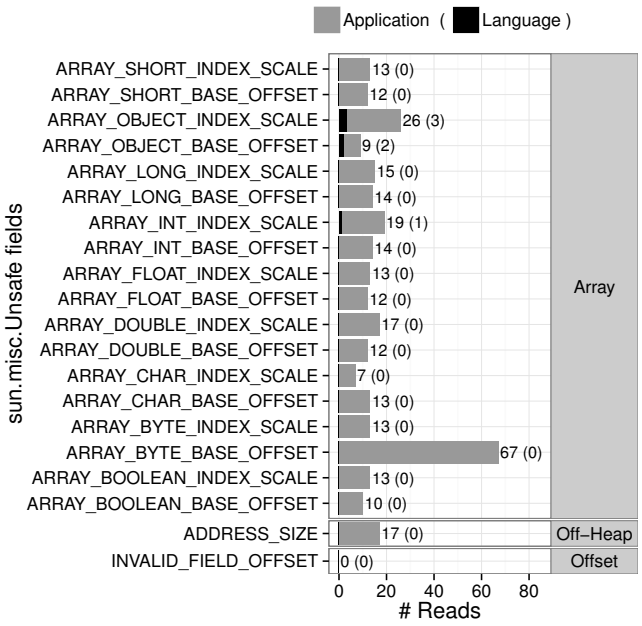

Figure 2. *sun.misc.Unsafe* field usage on Maven Central

instructions. These methods were introduced only recently in Java 8, which explains their limited use in our data set. We expect that their use will increase over time and that other operations, such as those in the *Ordered Put*, or *Volatile Put* groups will decrease as programmers use the lower-level fence operations.

- The *Fetch & Add* group, like the *CAS* group, allows the programmer to atomically update a Java variable. This group of methods was also added recently in Java 8. We expect their use to increase as programmers replace some calls to methods in the *CAS* group with the new functionality.
- The *Heap* group methods are used to directly access memory in the Java heap. The *Heap Get* and *Heap Put* groups allow the developer to load and store a Java variable. These groups are among the most frequently used ones in *Unsafe*.
- The *Misc* group contains the method *getLoadAverage*, to get the load average in the operating system run queue assigned to the available processors. It is not used.
- The *Monitor* group contains methods to explicitly manage Java monitors. The *tryMonitorEnter* method is never used.
- The *Off-Heap* group provides access to unmanaged memory, enabling explicit memory management. Similarly to the *Heap Get* and *Heap Put* groups, the *Off-Heap Get* and *Off-Heap Put* groups allow the developer to load and store values in Off-Heap memory. The usage of these methods is non-negligible, with *getByte* and *putByte* dominating the rest. The value of the *AD-*

DRESS_SIZE field is the result of the method *addressSize()*.

- Methods of the *Offset* group are used to compute the location of fields within Java objects. The offsets are used in calls to many other *sun.misc.Unsafe* methods, for instance those in the *Heap Get*, *Heap Put*, and the *CAS* groups. The method *objectFieldOffset* is the most called method in *sun.misc.Unsafe* due to its result being used by many other *sun.misc.Unsafe* methods. The *fieldOffset* method is deprecated, and indeed, we found no uses. The *INVALID_FIELD_OFFSET* field indicates an invalid field offset; it is never used because code using *objectFieldOffset* is not written in a defensive style.
- The *Ordered Put* group has methods to store to a Java variable without emitting any memory barrier but guaranteeing no reordering across the store.
- The *park* and *unpark* methods are contained in the *Park* group. With them, it is possible to block and unblock a thread's execution.
- The *throwException* method is contained in the *Throw* group, and allows one to throw checked exceptions without declaring them in the `throws` clause.
- Finally, the *Volatile Get* and *Volatile Put* groups allow the developer to store a value in a Java variable with `volatile` semantics.

It is interesting to note that despite our large corpus of code, there are several Unsafe methods that are never actually called. If *Unsafe* is to be used in third-party code, then it might make sense to extract those methods into a separate class to be only used from within the runtime library.

5. Question/Answer Database Analysis

To complement the analysis of *sun.misc.Unsafe* usage in practice, and to answer the questions relative to which features are commonly used ($Q2$), why they are used ($Q3$), and if they generate issues or problems ($Q4$), we search for discussions concerning the usage of *sun.misc.Unsafe* in Stack Overflow. We cannot rely only on Stack Overflow discussion tags: The topic is rarely discussed, and the tag *unsafe* is too general. A more precise analysis of the discussion contents is required to understand if it actually involves *sun.misc.Unsafe*. We rely on parsing *island grammars* [17] of structured fragments in natural language artifacts [2, 20] to discover discussions that involve *sun.misc.Unsafe* from the Stack Overflow data dump of March 2015[18].

5.1 (Island) Parsing Stack Overflow Discussions

Our island grammar parsing approach can be used to discover and analyze specific constructs that reveal the usage of *sun.misc.Unsafe*. For example, a discussion could report a code sample using *sun.misc.Unsafe*, or a user could mention the class (or one of its fields/methods) in an answer to a question concerning some specific problem that the usage of *sun.misc.Unsafe* can tackle. The island grammar allows us to identify constructs like stack traces and Java code fragments, including (potentially incomplete) method invocations or mentions inside natural language text. We do not perform simple identification and extraction, but we rather model the contents with a Heterogeneous Abstract Syntax Tree (H-AST) [20].

Identifying Relevant Discussions: We identify Stack Overflow discussions concerning the *sun.misc.Unsafe* class by analyzing all the discussions tagged with one of *java*, *scala*, *android*, or *jvm*. To understand if a discussion concerns *sun.misc.Unsafe*, we search for (i) uses of fields or methods exposed by *Unsafe* or (ii) mentions of the class itself. We extract discussions where the HAST contains either (i) a qualified identifier matching *Unsafe*, *unsafe*, *UNSAFE*, or *sun.misc.Unsafe*; (ii) an invocation of a method declared in *sun.misc.Unsafe*; or (iii) Java identifiers, beginning with a lowercase letter and containing a case change (e.g., "fieldOffset"), that respect method naming conventions and match one of the methods declared in *Unsafe*.

Refining Parsing Results: The *park* method appears at the top of stack traces for idle threads. Since these occurrences do not represent an interesting usage of *Unsafe*, we therefore removed all discussions where the only usage of *Unsafe* is the *park* method occurring in a stack trace.

In total we collected 20,915 discussions, out of which 560 report the type, 20,426 mention a method matching the ones of *Unsafe*, and 5 mention an *Unsafe* field. However, if the presence of the type *Unsafe* guarantees that the discussion is likely about *sun.misc.Unsafe*, the lone presence of the method name does not guarantee that (e.g., methods like *getInt* can be found in classes like *java.nio.ByteBuffer*[19]). Only 49 discussions contain a method mention and the string "unsafe", which result (after manual inspection) in only 18 effective discussions concerning *Unsafe*. In the end, our dataset contains 578 discussions.

5.2 Findings on Stack Overflow Discussions

Figure 3 presents an overview of *sun.misc.Unsafe* method mentions in Stack Overflow. The mentions are presented by distinguishing whether they appear only in the question, only in the answer, or in both. The list of methods does not distinguish between overloaded variants. In fact, people often mention method names without formal or actual parameters. Thus, in many cases, to understand which is the overloaded alternative one would have to do a manual inspection.

Discussions Archetypes. We manually inspected the resulting discussions to understand how *sun.misc.Unsafe* is discussed, devising a set of discussion archetypes:

[18] `https://archive.org/details/stackexchange`

[19] `http://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html`

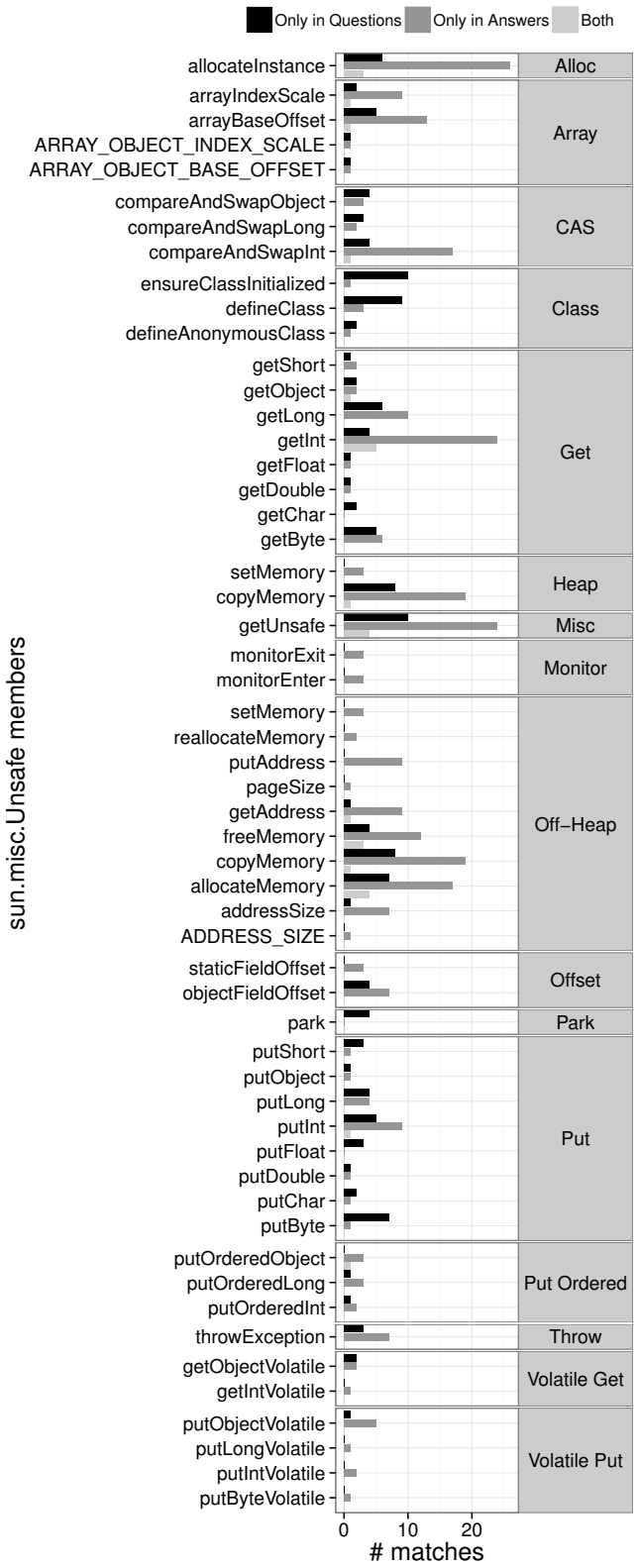

Figure 3. *sun.misc.Unsafe* members mentions on Stack Overflow

- **Lack of documentation:** Unsafe is an undocumented API, and a primary archetype concerns developers asking the crowd to obtain clarification on usage. A relatively popular question, entitled "Using sun.misc.Unsafe in real world", asks for typical use cases.[20]
- **Performance:** Users coming from unmanaged languages like C and C++ discuss how to avoid the cost of Java's safety checks. For instance, a user asked for an equivalent method call for *memcpy*.[21]
- **Misdirected uses:** Developers may propose *Unsafe* for inappropriate purposes. For example, a post discusses the use of the address to free an object on the Java heap.[22] Overall, the availability of Unsafe to developers who do not have a deep understanding of the JVM comes with the risk of misdirected uses. This risk is not unlike the risk of inappropriately using eval [23] in JavaScript.

6. Finding *sun.misc.Unsafe* Usage Patterns

We examined the artifacts in the Maven Central software repository to identify usage patterns for *Unsafe*. This section describes our methodology for identifying these patterns.

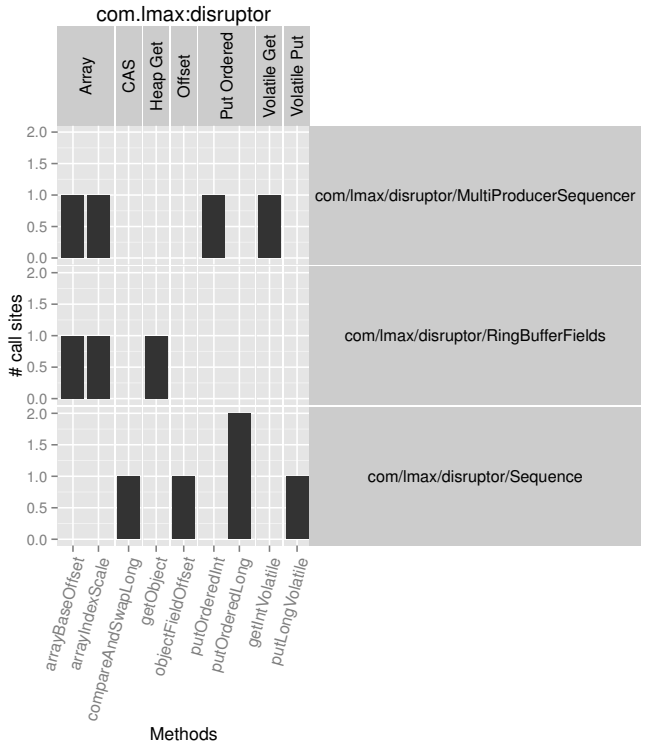

Figure 4. *com.lmax:disruptor* call sites

Our first step is to visualize how an artifact uses *Unsafe*. To this end, we count the *Unsafe* call sites and field usages per class in each artifact. Figures 4 and 5 show two examples

[20] http://stackoverflow.com/questions/5574241

[21] http://stackoverflow.com/questions/6060163

[22] http://stackoverflow.com/questions/24429777

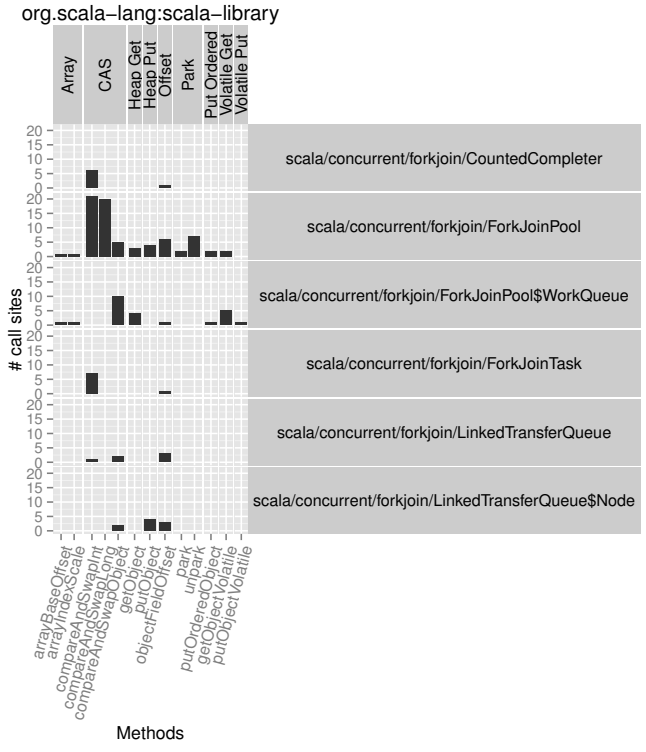

Figure 5. *org.scala-lang*:*scala-library* call sites

of call sites usages for *com.lmax*:*disruptor* and *org.scala-lang*:*scala-library* respectively. Each row shows a fully qualified class name and their usage of *sun.misc.Unsafe*.

After determining the call sites and field usage per artifact, we tried to find a way to group artifacts by how they use *sun.misc.Unsafe*. The first issue is to determine which method calls work together to achieve a goal. These calls might all be located within a single class, be spread across different classes within a package, or be spread across different packages within the whole artifact. After trying different combinations, we decided to group together calls occurring within a single class and its inner classes.

We cluster classes and their inner classes by *Unsafe* method usage using a dendrogram. Because a dendrogram can result in different clusters depending on at which height the dendrogram is cut, we experimented with various clusterings until settling on 31 clusters. An example of a cluster and its dendrogram is shown in Figure 6. In the figure we can see classes using methods of the *Off-Heap*, *Off-Heap Get*, and *Off-Heap Put* groups to implement large arrays.

Once we had a clustering of the artifacts by method usage, we manually inspected a sample of artifacts in each cluster to identify patterns. Some artifacts contained more than one pattern. For instance the cluster in Figure 6 contains classes that use *Unsafe* to implement large off-heap arrays, but also contains calls to methods of the *Put Volatile* group used to implement strongly shared consistent variables. We

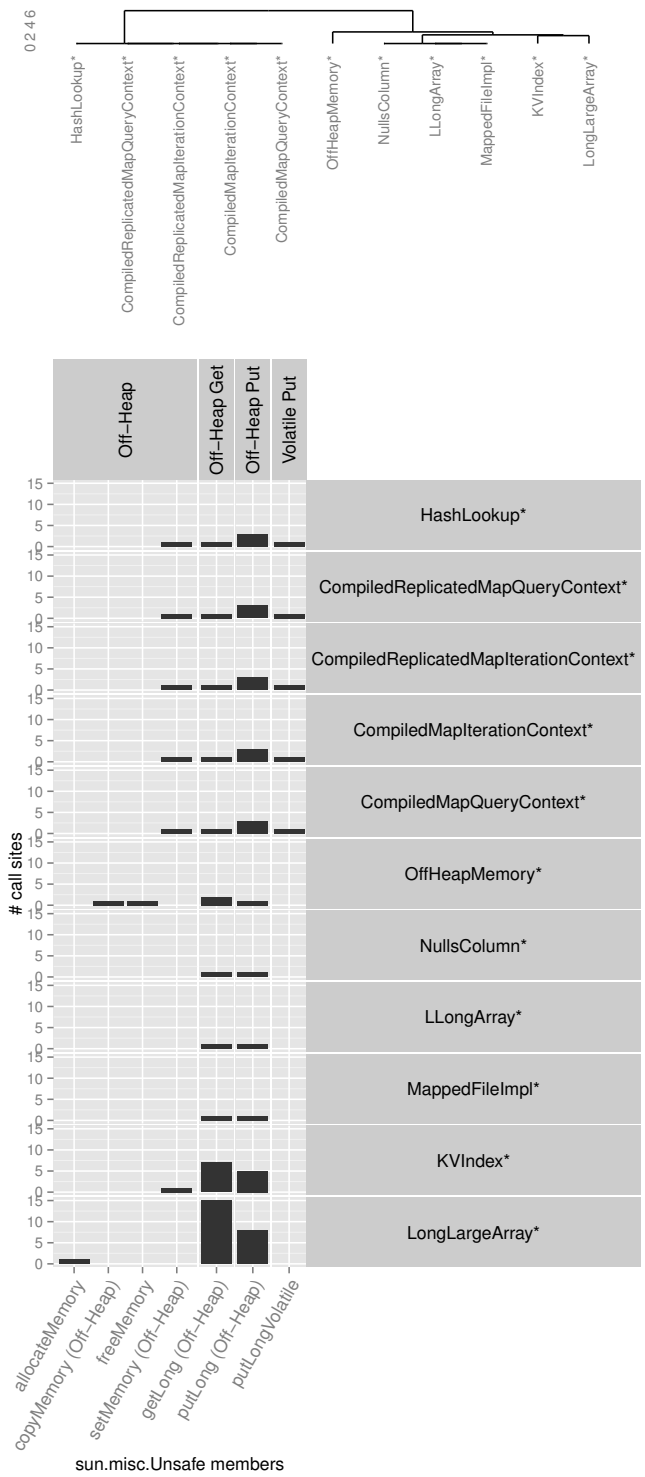

Figure 6. Classes using off-heap large arrays

tagged each artifact manually inspected with the set of patterns that it exhibits.

7. Usage Patterns of *sun.misc.Unsafe*

This section presents the patterns we have found during our study. We present them sorted by how many artifacts depend on them, as computed from the Maven dependency graph described in Section 4.

A summary of the patterns is shown in Table 1. The ***Pattern*** column indicates the name of the pattern. ***Found in*** indicates the number of artifacts in Maven Central that contain the pattern. ***Used by*** indicates the number of artifacts that transitively depend on the artifacts with the pattern. ***Most used artifacts*** presents the three most used artifacts containing the pattern, that is, the artifact with the most other artifacts that transitively depend upon it. Artifacts are shown using their Maven identifier, i.e., ⟨*groupId*⟩:⟨*artifactId*⟩.

We present each pattern using the following template.

Description. What is the purpose of the pattern? What does it do?
Rationale. What problem is the pattern trying to solve? In what contexts is it used?
Implementation. How is the pattern implemented using *sun.misc.Unsafe*?
Issues. Issues to consider when using the pattern and problems discussed in the Stack Overflow database.

7.1 Allocate an object without invoking a constructor

Description. With this pattern an object can be allocated on the heap without executing its constructor.
Rationale. This pattern is useful for creating mock objects for testing and in deserializing serialized objects.
Implementation. The *allocateInstance* method takes as parameter a *java.lang.Class* object, and returns a new instance of that class. Unlike allocating an object directly, or through the reflection API, the object's constructor is not invoked.
Issues. If the constructor is not invoked, the object might be left uninitialized and its invariants might not hold. Users of *allocateInstance* must take care to properly initialize the object before it is used by other code. This is often done in conjunction with other methods of *Unsafe*, for instance those in the *Heap Put* group, or by using the Java reflection API.

7.2 Process byte arrays in block

Description. When processing the elements of a byte array, better performance can be achieved by processing the elements 8 bytes at a time, treating it as a long array, rather than one byte at a time.
Rationale. The pattern is used for fast byte array processing, for instance, when comparing two byte arrays lexicographically.
Implementation. The *arrayBaseOffset* method is invoked to get the base offset of the byte array. Then *getLong* is used to fetch and process 8 bytes of the array at a time.
Issues. The pattern assumes that bytes in an array are stored contiguously. This may not be true for some VMs, e.g., those implementing large arrays using discontinuous arrays or arraylets [3, 30]. Users of the pattern should be aware of the endianness of the underlying hardware. In one Stack Overflow discussion, this pattern is discouraged since it is non-portable and, on many JVMs, results in slower code[23].

7.3 Atomic operations

Description. To implement non-blocking concurrent data structures and synchronization primitives, hardware-specific atomic operations provided by *sun.misc.Unsafe* are used.
Rationale. Non-blocking algorithms often scale better than algorithms that use locking.
Implementation. To get the offset of a Java variable either *objectFieldOffset* or *arrayBaseOffset*/*arrayIndexScale* can be used. With this offset, the methods from the *CAS* or *Fetch & Add* groups are used to perform atomic operations on the variable. Other methods of *Unsafe* are often used in the implementation of concurrent data structures, including *Volatile Get/Put*, *Ordered Put*, and *Fence* methods.
Issues. Non-blocking algorithms can be difficult to implement correctly. Programmers must understand the Java memory model and how the *Unsafe* methods interact with the memory model.

7.4 Strongly consistent shared variables

Description. Because of Java's weak memory model, when implementing concurrent code, it is often necessary to ensure that writes to a shared variable by one thread become visible to other threads, or to prevent reordering of loads and stores. Volatile variables can be used for this purpose, but *sun.misc.Unsafe* can be used instead with better performance. Additionally, because Java does not allow array elements to be declared volatile, there is no possibility other than to use *Unsafe* to ensure visibility of array stores. The methods of the *Ordered Put* groups and the *Volatile Get/Put* groups can be used for these purposes. In addition, the *Fence* methods were introduced in Java 8 expressly to provide greater flexibility for this use case.
Rationale. This pattern is useful for implementing concurrent algorithms or shared variables in concurrent settings. For instance, JRuby uses a *fullFence* to ensure visibility of writes to object fields.
Implementation. To ensure a write is visible to another thread, *Volatile Put* methods or *Ordered Put* methods can be used, even on non-volatile variables. Alternatively, a *storeFence* or *fullFence* can be used. *Volatile Get* methods ensure other loads and stores are not reordered across the load. A *loadFence* could also be used before a read of a shared variable.
Issues. Fences can replace volatile variables in some situations, offering better performance. Most of the uses of the

[23] `http://stackoverflow.com/questions/12226123`

Table 1. Patterns and their occurrences in the Maven Central repository.

	Pattern	Found In	Used by	Most used artifacts
1	Allocate an object without invoking a constructor	88	14794	*org.springframework*:*spring-core*, *org.objenesis*:*objenesis*, *org.mockito*:*mockito-all*
2	Process byte arrays in block	44	12274	*com.google.guava*:*guava*, *com.google.gwt*:*gwt-dev*, *net.jpountz.lz4*:*lz4*
3	Atomic operations	84	10259	*org.scala-lang*:*scala-library*, *org.apache.hadoop*:*hadoop-hdfs*, *org.glassfish.grizzly*:*grizzly-framework*
4	Strongly consistent shared variables	198	9795	*org.scala-lang*:*scala-library*, *org.jruby*:*jruby-core*, *com.hazelcast*:*hazelcast-all*
5	Park/unpark Threads	62	7330	*org.scala-lang*:*scala-library*, *org.codehaus.jsr166-mirror*:*jsr166y*, *com.netflix.servo*:*servo-internal*
6	Update final fields	11	7281	*org.codehaus.groovy*:*groovy-all*, *org.jodd*:*jodd-core*, *com.lmax*:*disruptor*
7	Non-lexically-scoped monitors	14	7015	*org.jboss.modules*:*jboss-modules*, *org.apache.cassandra*:*cassandra-all*, *org.gridgain*:*gridgain-core*
8	Serialization/deserialization	32	5689	*com.hazelcast*:*hazelcast-all*, *com.esotericsoftware.kryo*:*kryo*, *com.thoughtworks.xstream*:*xstream*
9	Foreign data access and object marshaling	8	3690	*eu.stratosphere*:*stratosphere-core*, *com.github.jnr*:*jffi*, *org.python*:*jython*
10	Throw checked exceptions without being declared	59	3566	*io.netty*:*netty-all*, *net.openhft*:*lang*, *ai.h2o*:*h2o-core*
11	Get the size of an object or an array	4	3003	*net.sf.ehcache*:*ehcache*, *com.github.jbellis*:*jamm*, *org.openjdk.jol*:*jol-core*
12	Large arrays and off-heap data structures	12	487	*org.neo4j*:*neo4j-primitive-collections*, *com.orientechnologies*:*orientdb-core*, *org.mapdb*:*mapdb*
13	Get memory page size	11	359	*org.apache.hadoop*:*hadoop-common*, *net.openhft*:*lang*, *org.xerial.larray*:*larray-mmap*
14	Load class without security checks	21	294	*org.elasticsearch*:*elasticsearch*, *org.apache.geronimo.ext.openejb*:*openejb-core*, *net.openhft*:*lang*

pattern use the *Ordered Put* and *Volatile Put* methods. Since they were added to Java only recently, there are currently few instances of the pattern that use the *Fence* methods.

7.5 Park/unpark Threads

Description. The *park* and *unpark* methods are used to block and unblock threads and are useful for implementing locks and other blocking synchronization constructs.

Rationale. The alternative to parking a thread is to busy-wait, which uses CPU resources and does not allow other threads to proceed.

Implementation. The *park* method blocks the current thread while *unpark* unblocks a thread given as an argument.

Issues. Users of *park* must be careful to avoid deadlock.

7.6 Update final fields

Description. This pattern is used to update a final field.

Rationale. Although it is possible to use reflection to implement the same behavior, updating a final field is easier and more efficient using *sun.misc.Unsafe*. Some applications update final fields when cloning objects or when deserializing objects.
Implementation. The *objectFieldOffset* methods and one of the *Put* methods work in conjunction to directly modify the memory where a final field resides.
Issues. There are numerous security and safety issues with modifying final fields. The update should be done only on newly created objects (perhaps also using *allocateInstance* to avoid invoking the constructor) before the object becomes visible to other threads. The Java Language Specification (Section 17.5.3) [10] recommends that final fields not be read until all updates are complete. In addition, the language permits compiler optimizations with final fields that can prevent updates to the field from being observed. Since final fields can be cached by other threads, one instance of the pattern uses *putObjectVolatile* to update the field rather than simply *putObject*. Using this method ensures that any cached copy in other threads is invalidated.

7.7 Non-lexically-scoped monitors

Description. In this pattern, monitors are explicitly acquired and released without using `synchronized` blocks.
Rationale. The pattern is used in some situations to avoid deadlock, releasing a monitor temporarily, then reacquiring it.
Implementation. One usage of the pattern is to temporarily release monitor locks acquired in client code (e.g., through a synchronized block or method) and then to reenter the monitor before returning to the client. The *monitorExit* method is used to exit the synchronized block. Because monitors are reentrant, the pattern uses the method *Thread.holdsLock* to implement a loop that repeatedly exits the monitor until the lock is no longer held. When reentering the monitor, *monitorEnter* is called the same number of times as *monitorExit* was called to release the lock.
Issues. Care must be taken to balance calls to *monitorEnter* and *monitorExit*, or else the lock might not be released or an `IllegalMonitorStateException` might be thrown.

7.8 Serialization/deserialization

Description. In this pattern, *sun.misc.Unsafe* is used to persist and subsequently load objects to and from secondary memory dynamically. Serialization in Java is so important that it has a *Serializable* interface to automatically serialize objects that implement it. Although this kind of serialization is easy to use, it does not offer good performance and is inflexible. It is possible to implement serialization using the reflection API. This is also expensive in terms of performance. Therefore, fast serialization frameworks often use *Unsafe* to get and set fields of objects. Some of these projects use reflection to check if *sun.misc.Unsafe* is available, falling back on a slower implementation if not.
Rationale. De/serialization requires reading and writing fields to save and restore objects. Some of these fields may be final or private.
Implementation. Methods of *Heap Get* and *Heap Put* are used to read and write fields and array elements. Deserialization may use *allocateInstance* to create objects without invoking the constructor.
Issues. Using *Unsafe* for serialization and deserialization has many of the same issues as using *Unsafe* for updating final fields (Section 7.6) and for creating objects without invoking a constructor (Section 7.1). Objects must not escape before being completely deserialized. Type safety can be violated by using methods of the *Heap Put* group. In addition, care must be taken when deserializing some data structures. For instance, data structures that use *System.identityHashCode* or *Object.hashCode* may need to rehash objects on deserialization because the deserialized object might have a different hash code than the original serialized object.

7.9 Foreign data access and object marshaling

Description. In this pattern *sun.misc.Unsafe* is used to share data between Java code and code written in another language, usually C or C++.
Rationale. This pattern is needed to efficiently pass data, especially structures and arrays, back and forth between Java and native code. Using this pattern can be more efficient than using native methods and JNI.
Implementation. The methods of the *Off-Heap* group are used to access memory off the Java heap. Often a buffer is allocated using *allocateMemory*, which is then passed to the other language using JNI. Alternatively, the native code can allocate a buffer in a JNI method. The *Off-Heap Get* and *Off-Heap Put* methods are used to access the buffer.
Issues. Use of *Unsafe* here is inherently not type-safe. Care must be taken especially with native pointers, which are represented as `long` values in Java code.

7.10 Throw checked exceptions without being declared

Description. This pattern allows the programmer to throw checked exceptions without being declared in the method's `throws` clause.
Rationale. In testing and mocking frameworks, the pattern is used to circumvent declaring the exception to be thrown, which is often unknown. It is used in the Java Fork/Join framework to save the generic exception of a thread to be rethrown later.
Implementation. The pattern is implemented using the *throwException* method.
Issues. This method can violate Java's subtyping relation, because it is not expected for a method that does not declare an exception to actually throw it. At run time, this can manifest as an uncaught exception.

7.11 Get the size of an object or an array

Description. This pattern uses *sun.misc.Unsafe* to estimate the size of an object or an array in memory.
Rationale. The object size can be useful for making manual memory management decisions. For instance, when implementing a cache, object sizes can be used to implement code to limit the cache size.
Implementation. To compute the size of an array, add *arrayBaseOffset* and *arrayIndexScale* (for the given array base type) times the array length. For objects, use *objectFieldOffset* to compute the offset of the last instance field. In both cases, a VM-dependent fudge factor is added to account for the object header and for object alignment and padding.
Issues. Object size is very implementation dependent. Accounting for the object header and alignment requires adding VM-dependent constants for these parameters.

7.12 Large arrays and off-heap data structures

Description. This pattern uses off-heap memory to create large arrays or data structures with manual memory management.
Rationale. Java's arrays are indexed by `int` and are thus limited to 2^{31} elements. Using *Unsafe*, larger buffers can be allocated outside the heap.
Implementation. A block of memory is allocated with *allocateMemory* and then accessed using *Off-Heap Get* and *Off-Heap Put* methods. The block is freed with *freeMemory*.
Issues. This pattern has all the issues of manual memory management: memory leaks, dangling pointers, double free, etc. One issue, mentioned on Stack Overflow, is that the memory returned by *allocateMemory* is uninitialized and may contain garbage.[24] Therefore, care must be taken to initialize allocated memory before use. The *Unsafe* method *setMemory* can be used for this purpose.

7.13 Get memory page size

Description. *sun.misc.Unsafe* is used to determine the size of a page in memory.
Rationale. The page size is needed to allocate buffers or access memory by page. A common use case is to round up a buffer size, typically a *java.nio.ByteBuffer*, to the nearest page size. Hadoop uses the page size to track memory usage of cache files mapped directly into memory using *java.nio.MappedByteBuffer*. Another use is to process a buffer page-by-page. Some native libraries require or recommend allocating buffers on page-size boundaries.[25]
Implementation. Call *pageSize*.
Issues. Some platforms on which the JVM runs do not have virtual memory, so requesting the page size is non-portable.

[24] `http://stackoverflow.com/questions/16723244`

[25] `http://stackoverflow.com/questions/19047584`

7.14 Load class without security checks

Description. *sun.misc.Unsafe* is used to load a class from an array containing its bytecode. Unlike with the *ClassLoader* API, security checks are not performed.
Rationale. This pattern is useful for implementing lambdas, dynamic class generation, and dynamic class rewriting. It is also useful in application frameworks that do not interact well with user-defined class loaders.
Implementation. The pattern is implemented using the *defineClass* method, which takes a byte array containing the bytecode of the class to load.
Issues. The pattern violates the Java security model. Untrusted code could be introduced into the same protection domain as trusted code.

8. Discussion

Many of the patterns we found indicate that *Unsafe* is used to achieve better performance or to implement functionality not otherwise available in the Java language or standard library.

However, many of the patterns described can be implemented using APIs already provided in the Java standard library. In addition, there are several existing proposals to improve the situation with *Unsafe* already under development within the Java community. Oracle software engineer Paul Sandoz [28] performed a survey on the OpenJDK mailing list to study how Unsafe is used[26] and describes several of these proposals.

A summary of the patterns with existing and proposed alternatives to *Unsafe* is shown in Table 2. The table consists of the following columns: The ***Pattern*** column indicates the name of the pattern. The next three columns indicate whether the pattern could be implemented either as a language feature (***Lang***), virtual machine extension (***VM***), or library extension (***Lib***). The ***Ref*** column indicates that the pattern can be implemented using reflection. A bullet (•) indicates that an alternative exists in the Java language or API. A check mark (✓) indicates that there is a proposed alternative for Java.

Many Java APIs already exist that provide functionality similar to *Unsafe*. Indeed, these APIs are often implemented using *Unsafe* under the hood, but they are designed to be used safely. They maintain invariants or perform runtime checks to ensure that their use of *Unsafe* is safe. Because of this overhead, using *Unsafe* directly should in principle provide better performance at the cost of safety.

For example, the *java.util.concurrent* package provides classes for safely performing atomic operations on fields and array elements, as well as several synchronizer classes. These classes can be used instead of *Unsafe* to implement atomic operations (Section 7.3) or strongly consistent shared variables (Section 7.4). The standard library class *java.util.concurrent.locks.LockSupport* provides *park* and *unpark*

[26] `http://www.infoq.com/news/2014/02/Unsafe-Survey`

Table 2. Patterns and their alternatives. A bullet (•) indicates that an alternative exists in the Java language or API. A check mark (✓) indicates that there is a proposed alternative for Java.

	Pattern	Lang	VM	Lib	Ref
1	Allocate an object without invoking a constructor	✓			
2	Process byte arrays in block		✓		
3	Atomic operations			•	
4	Strongly consistent shared variables			✓	
5	Park/unpark Threads			•	
6	Update final fields				•
7	Non-lexically-scoped monitors	✓			
8	Serialization/deserialization	✓		•	•
9	Foreign data access and object marshaling	✓		•	
10	Throw checked exceptions without being declared	✓			
11	Get the size of an object or an array	✓		✓	
12	Large arrays and off-heap data structures	✓		✓	
13	Get memory page size	✓		✓	
14	Load class without security checks	✓		✓	

methods to be used for implementing locks. These methods are just thin wrappers around the *sun.misc.Unsafe* methods of the same name and could be used to implement the park pattern (Section 7.5). Java already supports serialization of objects (Section 7.8) using the *java.lang.Serializable* and *java.io.ObjectOutputStream* API. The now-deleted JEP 187 Serialization 2.0 proposal[27] [28] addresses some of the issues with Java serialization.

Because volatile variable accesses compile to code that issues memory fences, strongly consistent variables (Section 7.4) can be implemented by accessing volatile variables. However, the fences generated for volatile variables may be stronger (and therefore less performant) than are needed for a given application. Indeed, the *Unsafe Put Ordered* and *Fence* methods were likely introduced to improve performance versus volatile variables. There is currently a proposal for enhanced volatile support in the JVM (JEP 193 Enhanced Volatiles [15]). This proposal introduces *variable handles*, which allow atomic operations on fields and array elements.

Many of the patterns can be implemented using the reflection API, albeit with lower performance than with *Unsafe* [13]. For example, reflection can be used for accessing object fields to implement serialization (Section 7.8). Similarly, reflection can be used in combination with *java.nio.ByteBuffer* and related classes for data marshaling (Section 7.9). The reflection API can also be used to write to final fields (Section 7.6). However, this feature of the reflection API makes sense only during deserialization or during object construction and may have unpredictable behavior in other cases.[29] Writing a final field through reflection may not ensure the write becomes visible to other threads that might have cached the final field, and it may not work correctly at all if the VM performs compiler optimizations such as constant propagation on final fields.

Many patterns use *Unsafe* to use memory more efficiently. Using structs or packed objects can reduce memory overhead by eliminating object headers and other per-object overhead. Java has no native support for structs, but they can be implemented with byte buffers or with JNI.[30]

The Arrays 2.0 proposal [26] and the value types proposal [25] address the large arrays pattern (Section 7.12). Project Sumatra [19] proposes features for accessing GPUs and other accelerators, one of the use cases for foreign data access (Section 7.9). Related proposals include JEP 191 [18], which proposes a new foreign function interface for Java, and Project Panama [27], which supports native data access from the JVM.

A *sizeof* feature could be introduced into the language or into the standard library (Section 7.11). A use case for this feature includes cache management implementations. A higher level alternative might be to provide an API for memory usage tracking in the JVM. A page size (Section 7.13) method could be added to the standard library, perhaps in the *java.nio* package, which already includes *MappedByteBuffer* to access memory-mapped storage.

Other patterns may require Java language changes. For instance, the language could be changed to not require methods to declare the exceptions they throw, obviating the need for *Unsafe* (Section 7.10) in this case. Indeed, there is a long-running debate[31] about the software-engineering benefits of checked exceptions. C#, for instance, does not require that exceptions be declared in method signatures at all. One alternative not requiring a language change, proposed in a Stack Overflow discussion, is to use Java generics instead.[32] Because of type erasure, a checked exception can be coerced unsafely into an unchecked exception and thrown.

[27] http://mail.openjdk.java.net/pipermail/core-libs-dev/2014-January/024589.html

[28] http://web.archive.org/web/20140702193924/http://openjdk.java.net/jeps/187

[29] http://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Field.html#set(java.lang.Object,\%20java.lang.Object)

[30] http://www.oracle.com/technetwork/java/jvmls2013sciam-2013525.pdf

[31] http://www.ibm.com/developerworks/library/j-jtp05254/

[32] http://stackoverflow.com/questions/11410042

Changing the language to support allocation without constructors (Section 7.1) or non-lexically-scoped monitors (Section 7.7) is feasible. However, implemention of these features must be done carefully to ensure object invariants are properly maintained. In particular, supporting arbitrary unconstructed objects can require type system changes to prevent usage of the object before initialization [22]. Limiting the scope of this feature to support deserialization only may be a good compromise and has been suggested in the JEP 187 Serialization 2.0 proposal.

Since *Unsafe* is often used simply for performance reasons, virtual machine optimizations can reduce the need for *Unsafe*. For example, the JVM's runtime compiler can be extended with optimizations for vectorizing byte array accesses, eliminating the motivation to use *Unsafe* to process byte arrays (Section 7.2). Many patterns use *Unsafe* to use memory more efficiently. This could be ameliorated with lower GC overhead. There are proposals for this, for instance JEP 189 Shenandoah: Low Pause GC [5].

9. Related Work

Oracle software engineer Paul Sandoz performed some informal analysis of Maven artifacts and usages in Grepcode [29] and conducted a survey to study how *Unsafe* is used [28]. The survey consists of 7 questions[33] that help to understand what pieces of *sun.misc.Unsafe* should be mainstreamed. We go beyond Sandoz' work by performing a comprehensive study of the Maven Central software repository to analyze how and why *sun.misc.Unsafe* is being used.

In the remainder of this section we first describe the related work about mining software repositories to understand a specific language feature. Then, we show where *Unsafe* fits in the broader spectrum, i.e., how to do low-level coding in a high level language.

9.1 Mining repositories to assess language features

Several researchers have mined software repositories with the goal of analyzing and understanding if, how and when certain programming language features are being used.

Dyer et. al. [6] studied the adoption of Java language features over time. Richards et. al. [23] present an in-depth study on the `eval` function in JavaScript. Mayer et. al. [16] studied the impact of type systems on software development. Callaú et. al. [4] performed an empirical study to assess how much the dynamic and reflective features of Smalltalk are actually used in practice. Holkner and Harland [12] did a similar study on production-stage open source Python programs. Richards et. al. [24] also did a study on the dynamic behavior but applied to JavaScript programs. Gorla et. al. [9] mined a large set of Android applications, clustering applications by their description topics and identifying outliers in each cluster with respect to their API usage. Grechanik et. al. [11] also mined large scale software repositories to obtain several statistics on how source code is actually written.

[33] `http://www.infoq.com/news/2014/02/Unsafe-Survey`

9.2 High-level language semantics for low-level coding

Oracle provides the *sun.misc.Unsafe* class for low-level programming, e.g, synchronization primitives, direct memory access methods, array manipulation and memory usage. Although the *sun.misc.Unsafe* class is not officially documented, there is literature based on it.

Korland et. al. [13] presented a Java STM framework, intended as a development platform for scalable concurrent applications and as a research tool for designing STM algorithms. They chose to use *sun.misc.Unsafe* to implement fast reflection, as it proved to be vastly more efficient than the standard Java reflection mechanisms. Pukall et. al. [21] introduced a runtime update approach based on Java that offers flexible dynamic software updates with minimal performance overhead. They used the `allocateInstance` method, because it eases the creation of instances even if the class has no default constructor. Gligoric et. al. [8] proposed a new approach to serialization/deserialization via code generation, using *sun.misc.Unsafe* to allocate instances and to set the fields. The Jikes RVM [1] is a Java Virtual Machine targeting researchers in runtime systems. It is a Java-in-Java virtual machine because is itself built in Java, a style of implementation termed meta-circular. The Jikes RVM provides an implementation of *sun.misc.Unsafe* with the *magic* framework. Frampton et. al. [7] proposed `org.vmmagic` to provide an escape hatch to low-level alternatives needed to build virtual machines; however, they require compiler support.

10. Conclusions

sun.misc.Unsafe is an API that was designed for limited use in system-level runtime library code. The *Unsafe* API is powerful, but dangerous. The improper use of *Unsafe* undermines Java's safety guarantees. We studied to what degree *Unsafe* usage has spread into third-party libraries, to what degree such third-party usage of *Unsafe* can impact existing Java code, and which *Unsafe* API features such third-party libraries actually use. We studied the questions and discussions developers have about *Unsafe*, and we identified common usage patterns. We thereby provided a basis for evolving the *Unsafe* API, the Java language, and the JVM by eliminating unused or abused unsafe features, and by providing safer alternatives for features that are used in meaningful ways. We hope this will help to make *Unsafe* safer.

Acknowledgments

We thank the reviewers for their careful comments. Paul Sandoz gave us many useful and encouraging comments. Mastrangelo was supported by Swiss National Science Foundation grant CRSII2_136225. Ponzanelli and Lanza

were supported by Swiss National Science Foundation Project ESSENTIALS, No. 153129.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-source Research Community. *IBM Syst. J.*, 44(2):399–417, January 2005.

[2] Alberto Bacchelli, Anthony Cleve, Michele Lanza, and Andrea Mocci. Extracting structured data from natural language documents with island parsing. In *Proceedings of ASE 2011 (26th IEEE/ACM International Conference On Automated Software Engineering)*, pages 476–479, 2011.

[3] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 285–298, New York, NY, USA, 2003. ACM.

[4] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How developers use the dynamic features of programming languages: The case of Smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 23–32, New York, NY, USA, 2011. ACM.

[5] Roman Kennke Christine H. Flood. JEP 189: Shenandoah: An Ultra-Low-Pause-Time Garbage Collector. `http://openjdk.java.net/jeps/189`, 2014.

[6] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of AST nodes to study actual and potential usage of Java language features. In *36th International Conference on Software Engineering*, ICSE'14, pages 779–790, June 2014.

[7] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying Magic: High-level Low-level Programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM.

[8] Milos Gligoric, Darko Marinov, and Sam Kamin. CoDeSe: Fast Deserialization via Code Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 298–308, New York, NY, USA, 2011. ACM.

[9] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1025–1035, New York, NY, USA, 2014. ACM.

[10] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 2013.

[11] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 11:1–11:10, New York, NY, USA, 2010. ACM.

[12] Alex Holkner and James Harland. Evaluating the dynamic behaviour of Python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC '09, pages 19–28, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[13] Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive Concurrency with Java STM. In *Communications of the ACM, Invited Review Paper*, page 19 pages, 2010.

[14] Eugene Kuleshov. Using the ASM framework to implement common Java bytecode transformation patterns. In *Conference on Aspect Oriented Software Development (AOSD): Industry Track*, 2007.

[15] Doug Lea. JEP 193: Enhanced Volatiles. `http://openjdk.java.net/jeps/193`, 2014.

[16] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 683–702, New York, NY, USA, 2012. ACM.

[17] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of WCRE 2001 (8th Working Conference on Reverse Engineering)*, pages 13–22. IEEE CS, 2001.

[18] Charles Oliver Nutter. JEP 191: Foreign Function Interface. `http://openjdk.java.net/jeps/191`, 2014.

[19] OpenJDK. Project Sumatra. `http://openjdk.java.net/projects/sumatra/`, 2013.

[20] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. StORMeD: Stack Overflow ready made data. In *Proceedings of MSR 2015 (12th Working Conference on Mining Software Repositories)*, page to be published. ACM Press, 2015.

[21] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. JavAdaptor-Flexible runtime updates of Java applications. *Software: Practice and Experience*, 43(2):153–185, 2013.

[22] Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 53–65, New York, NY, USA, 2009. ACM.

[23] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN Confer-*

ence on Programming Language Design and Implementation, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

[25] John Rose, Brian Goetz, and Guy Steele. State of the Values. `http://cr.openjdk.java.net/~jrose/values/values-0.html`, 2014.

[26] John R. Rose. Arrays 2.0. `http://cr.openjdk.java.net/~jrose/pres/201207-Arrays-2.pdf`, 2012.

[27] John R. Rose. The isthmus in the VM. `https://blogs.oracle.com/jrose/entry/the_isthmus_in_the_vm`, 2014.

[28] Paul Sandoz. Safety not guaranteed: sun.misc.Unsafe and the quest for safe alternatives. `http://cr.openjdk.java.net/~psandoz/dv14-uk-paul-sandoz-unsafe-the-situation.pdf`, 2014. Oracle Inc. [Online; accessed 29-January-2015].

[29] Paul Sandoz. Personal communication, 2015.

[30] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '00, pages 9–17, New York, NY, USA, 2000. ACM.