# The **`sun.misc.Unsafe`** Situation

Paul Sandoz
Oracle

 @PaulSandoz

# Questions…

- What is **`sun.misc.Unsafe`**?
- Is it going away? If so when or where?
- What should be used instead?

**`sun.misc.Unsafe`** is an internal and unsafe abstraction in the JDK

for building better, faster and safer abstractions in the JDK

# Some stuff in unsafe is…

- Not necessarily all that unsafe (or perhaps used)
```
public native void throwException(Throwable ee);
```

- Not necessarily all that performant
```
@HotSpotIntrinsicCandidate
public native Object allocateInstance(Class<?> cls)
    throws InstantiationException;
```

- Hard to use optimally with HotSpot C1 and C2 compilers (JDK-8074124)
```
long o = (((long) i) << SCALE + OFFSET)
long v = Unsafe.getLong(base, o);
```

# Mr Aleksey Shipilev's metaphorical view* of **`Unsafe`** methods


* Image edited to protect the sensibilities of the public

http://www.inf.usi.ch/faculty/lanza/Downloads/Mast2015a.pdf

# 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 ca-

## 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]

http://www.inf.usi.ch/faculty/lanza/Downloads/Mast2015a.pdf

# 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


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

# Inside

Java reflection
Java serialization
NIO
java.util.concurrent
java.lang.invoke
CORBA performance
Crypto performance
JMX load average
Mac OS Objective C bridge
…

Sun microsystems

JDK Contents

---

# Why Developers Should Not Write Programs That Call 'sun' Packages

The classes that JavaSoft includes with the JDK fall into at least two packages: java.* and sun.*. Only classes in java.* packages are a standard part of the Java Platform and will be supported into the future. In general, API outside of java.* can change at any time without notice, and so cannot be counted on either across OS platforms (Sun, Microsoft, Netscape, Apple, etc.) or across Java versions. Programs that contain direct calls to the sun.* API are not 100% Pure Java. In other words:

**The java.* packages make up the official, supported, public Java interface.**
If a Java program directly calls only API in java.* packages, it will operate on all Java-compatible platforms, regardless of the underlying OS platform.

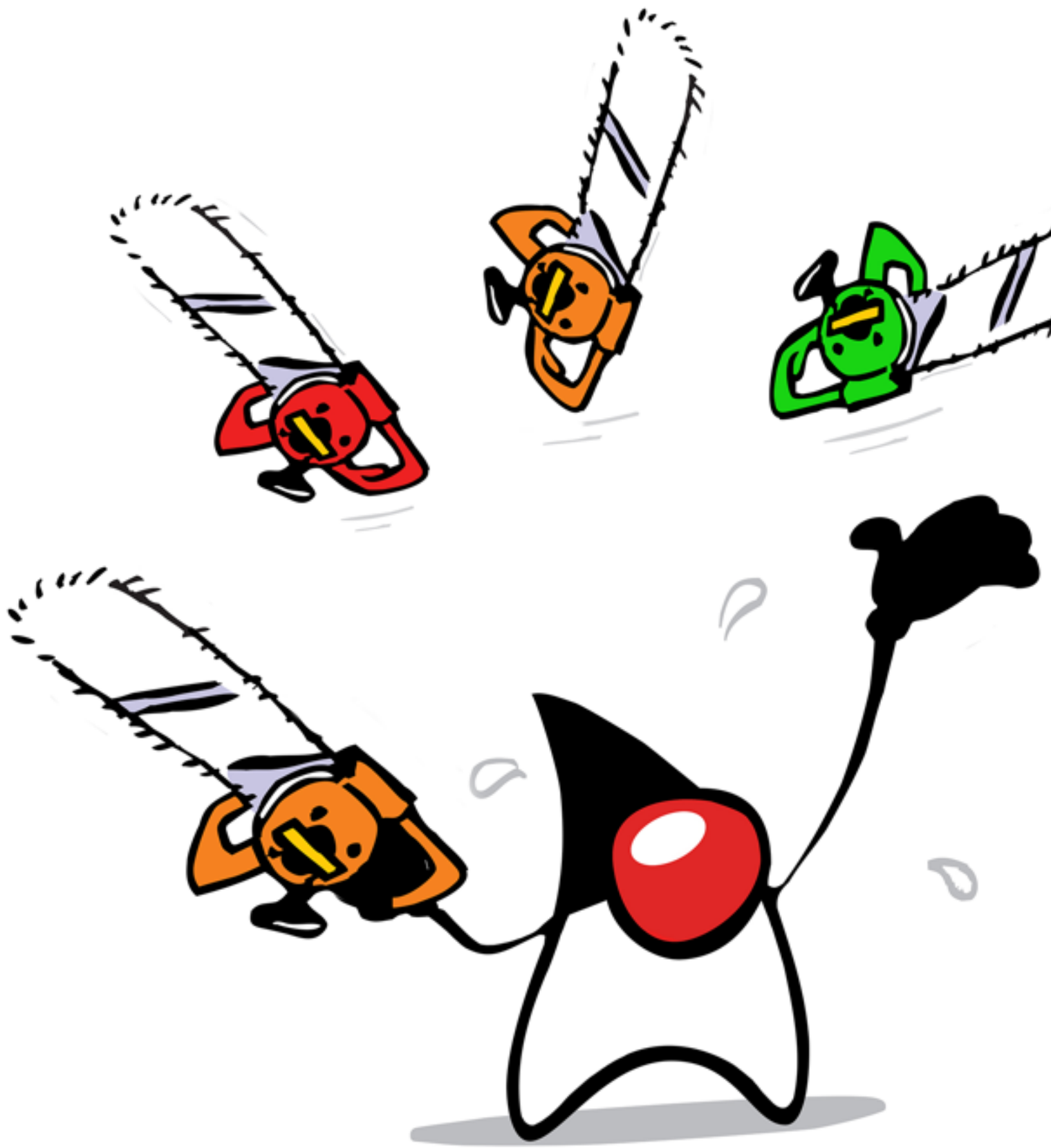
**The sun.* packages are *not* part of the supported, public Java interface.**
A Java program that directly calls any API in sun.* packages is *not* guaranteed to work on all Java-compatible platforms. In fact, such a program is not guaranteed to work even in future versions on the same platform.

For these reasons, there is no documentation available for the sun.* classes. Platform-independence is one of the great advantages of developing in Java. Furthermore, JavaSoft, and our licensees of Java technology, are committed to maintaining the APIs in java.* for future versions of the Java platform. (Except for code that relies on bugs that we later fix, or APIs that we deprecate and eventually remove.) This means that once your program is written, the binary will work in future releases. That is, future implementations of the java platform will be backward compatible.

Each company that implements the Java platform will do so in their own private way. The classes in sun.* are present in the JDK to support the JavaSoft implementation of the Java platform: the sun.* classes are what make the classes in java.* work "under the covers" for the JavaSoft JDK. These classes will not in general be present on another vendor's Java platform. If your Java program asks for a class "sun.package.Foo" by name, it will likely fail with ClassNotFoundError, and you will have lost a major advantage of developing in Java.

# Outside

Akka
Cassandra
Ehcache
Grails
Guava
HBase
Hadoop
Hazelcast
Hibernate
JRuby
Kafka
Kryo
LMAX Disruptor
Mockito
Neo4j
Netty
Objenesis
Scala
Solr
Spark
Spring
XStream
Zookeeper
...



stackoverflow

Questions | Tags | Users | Badge

# How to free memory using Java Unsafe, using a Java reference?

0

Java Unsafe class allows you to allocate memory for an object as follows, but using this method how would you free up the memory allocated when finished, as it does not provide the memory address...

```
    Field f = Unsafe.class.getDeclaredField("theUnsafe"); //Internal reference
    f.setAccessible(true);
    Unsafe unsafe = (Unsafe) f.get(null);

    //This creates an instance of player class without any initialization
    Player p = (Player) unsafe.allocateInstance(Player.class);
```

1

Is there a way of accessing the memory address from the object reference, maybe the integer returned by the default hashCode implementation will work, so you could do...

```
unsafe.freeMemory(p.hashCode());
```

doesn't seem right some how...

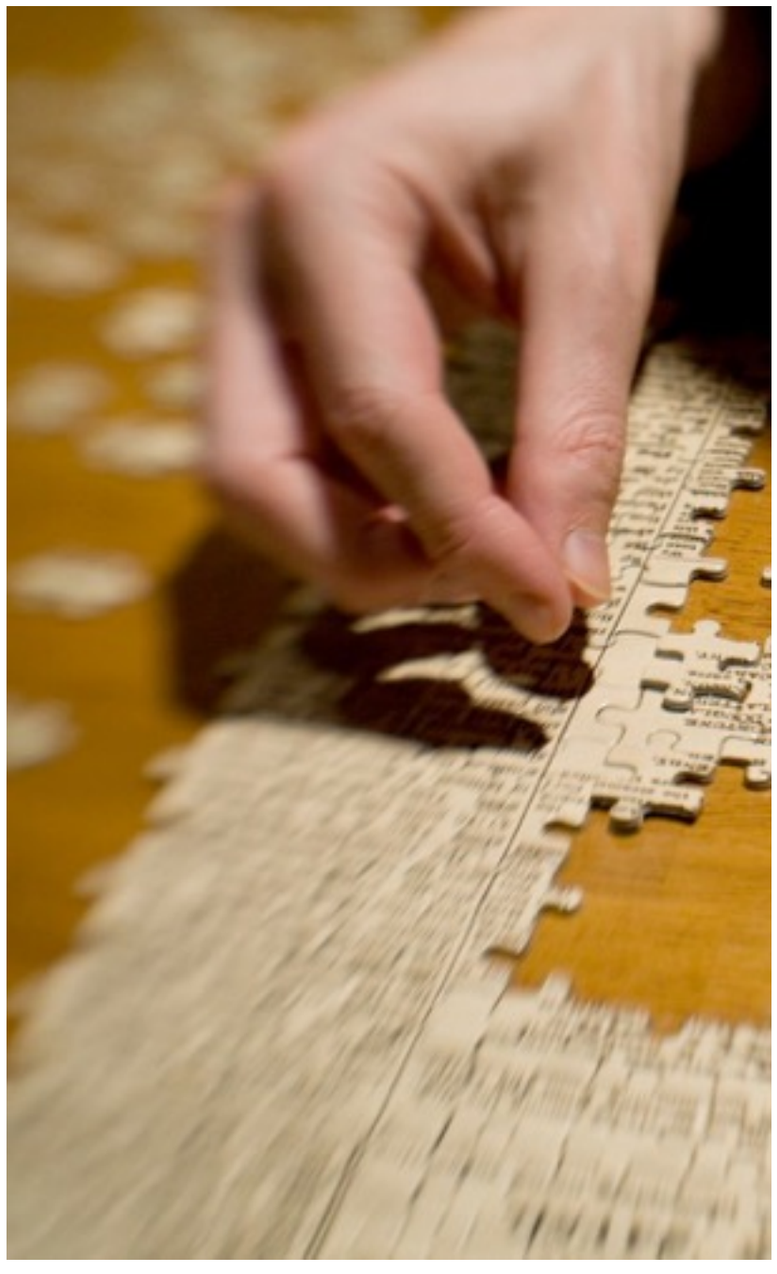
java | memory-management | jvm | directmemory

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00000001027a4b50, pid=59598, tid=0x0000000000001903
#
# JRE version: OpenJDK Runtime Environment (9.0) (build 1.9.0-internal)
# Java VM: OpenJDK 64-Bit Server VM (1.9.0-internal mixed mode bsd-amd64 compressed oops)
# Problematic frame:
# V  [libjvm.dylib+0x5a4b50]  Unsafe_SetNativeAddress+0x50
#
```

# In legal safety

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00000001027a4b50, pid=59598, tid=0x0000000000001903
#
# JRE version: OpenJDK Runtime Environment (9.0) (build 1.9.0-internal)
# Java VM: OpenJDK 64-Bit Server VM (1.9.0-internal mixed mode bsd-amd64 compressed oops)
# Problematic frame:
# V  [libjvm.dylib+0x5a4b50]  Unsafe_SetNativeAddress+0x50
#
# The following and previous is intended to outline our general product direction.
# It is intended for information purposes only, and may not be incorporated into
# any contract. It is not a commitment to deliver any material, code, or
# functionality, and should not be relied upon in making purchasing decisions.
# The development, release, and timing of any features or functionality
# described for Oracle's products remains at the sole discretion of Oracle.
```

# Java modularity enforcing strong boundaries



# JEP 260

# The fate of **`sun.misc.Unsafe`**

- Cloned and encapsulated in **`jdk.internal.*`** with JDK dependencies updated
- Available publicly
  - To code on the classpath (as is the case today)
  - To code in modules on request
- Will **not** be further enhanced

# JEP 260
# In JDK **$N**

- Replacement for a method exists in JDK **$N-1**
  ⇒ Encapsulate or remove that method in JDK **$N**

- Replacement for a method exists in JDK **$N**
  ⇒ Deprecate that method in JDK **$N**

- Replacements for all methods exist in JDK **$N-1**
  ⇒ Remove **sun.misc.Unsafe** in JDK **$N**

# Use cases

| Use case | Example methods |
|---|---|
| Concurrency primitives | **`Unsafe.compareAndSwap*`** |
| Serialization | **`Unsafe.allocateInstance`** (`ReflectionFactory.newConstructorForSerialization`) |
| Efficient memory management, layout, and access | **`Unsafe.allocate`**/**`freeMemory`** **`Unsafe.get*`**/**`put*`** (and JNI) |
| Interoperate across the JVM boundary | **`Unsafe.get*`**/**`put*`** (and JNI) |

# Use cases to features

| Use-case | Feature |
|---|---|
| Concurrency primitives | JEP 193 Variable Handles<br>x86 PAUSE spin loop hint |
| Serialization | Reboot JEP 187 Serialization Improvements |
| Efficient memory management, layout, and access | Project Panama, Project Valhalla, Arrays 2.0, Better GC |
| Interoperate across the JVM boundary | Project Panama,<br>JEP 191 Foreign Function Interface |

# Unsafe business as usual

- Projects Valhalla and Panama will significantly expand “close to the metal” support in the JDK

- New unsafe abstractions will inevitably appear for use in privileged JDK code

- As those unsafe abstractions mature new safer abstractions will be built on top

# Meanwhile… **`Unsafe`** is clearly insufficient for some use-cases


https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html


https://flink.apache.org/news/2015/09/16/off-heap-memory.html

# In the interim…


```
-XaddExports:j                                    s=ALL-UNNAMED
```

# Near term focus for Java 9

| Use-case | Feature |
| --- | --- |
| Concurrency primitives | JEP 193 Variable Handles |
| Serialization | |
| Efficient memory management, layout, and access | Improvements to array bounds checks and array access |
| Interoperate across the JVM boundary | |

# Array access

- Frameworks such as Guava, Hadoop, HBase and Avro use **`sun.misc.Unsafe`** for lexicographical **`byte[]`** comparison

- Update and add new methods in **`Arrays`** with optimized implementations

```
compareUnsigned(byte[], byte[])
compareUnsigned(byte[], int, int, byte[], int, int)
```

```
compareUnsigned(byte[], byte[])
compareUnsigned(byte[], int, int, byte[], int, int)
compareUnsigned(int[], int[])
compareUnsigned(int[], int, int, int[], int, int)
compareUnsigned(long[], long[])
compareUnsigned(long[], int, int, long[], int, int)
compareUnsigned(short[], short[])
compareUnsigned(short[], int, int, short[], int, int)

compare(boolean[], boolean[])
compare(boolean[], int, int, boolean[], int, int)
compare(byte[], byte[])
compare(byte[], int, int, byte[], int, int)
compare(char[], char[])
compare(char[], int, int, char[], int, int)
compare(double[], double[])
compare(double[], int, int, double[], int, int)
compare(float[], float[])
compare(float[], int, int, float[], int, int)
compare(int[], int[])
compare(int[], int, int, int[], int, int)
compare(long[], long[])
compare(long[], int, int, long[], int, int)
compare(short[], short[])
compare(short[], int, int, short[], int, int)
compare(T[], T[])
compare(T[], int, int, T[], int, int)
compare(T[], T[], Comparator)
compare(T[], int, int, T[], int, int, Comparator)
```

```
equals(boolean[], int, int, boolean[], int, int)
equals(byte[], int, int, byte[], int, int)
equals(char[], int, int, char[], int, int)
equals(double[], int, int, double[], int, int)
equals(float[], int, int, float[], int, int)
equals(int[], int, int, int[], int, int)
equals(long[], int, int, long[], int, int)
equals(short[], int, int, short[], int, int)
equals(Object[], int, int, Object[], int, int)
mismatch(boolean[], boolean[])
mismatch(boolean[], int, int, boolean[], int, int)
mismatch(byte[], byte[])
mismatch(byte[], int, int, byte[], int, int)
mismatch(char[], char[])
mismatch(char[], int, int, char[], int, int)
mismatch(double[], double[])
mismatch(double[], int, int, double[], int, int)
mismatch(float[], float[])
mismatch(float[], int, int, float[], int, int)
mismatch(int[], int[])
mismatch(int[], int, int, int[], int, int)
mismatch(long[], long[])
mismatch(long[], int, int, long[], int, int)
mismatch(short[], short[])
mismatch(short[], int, int, short[], int, int)
mismatch(Object[], Object[])
mismatch(Object[], int, int, Object[], int, int)
mismatch(T[], T[], Comparator)
mismatch(T[], int, int, T[], int, int, Comparator)
```

# The obvious implementation

```
public static int compareUnsigned(byte[] a, byte[] b) {
    if (a == b)
        return 0;
    if (a == null || b == null)
        return a == null ? -1 : 1;

    int length = Math.min(a.length, b.length);
    for (int i = 0; i < length; i++) {
        if (a[i] != b[i]) return Byte.compareUnsigned(a[i], b[i]);
    }

    return a.length - b.length;
}
```

# Using mismatch

```
public static int compareUnsigned(byte[] a, byte[] b) {
    if (a == b)
        return 0;
    if (a == null || b == null)
        return a == null ? -1 : 1;

    int i = ArraysSupport.mismatch(a, b,
                                   Math.min(a.length, b.length));
    if (i >= 0) {
        return Byte.compareUnsigned(a[i], b[i]);
    }

    return a.length - b.length;
}
```

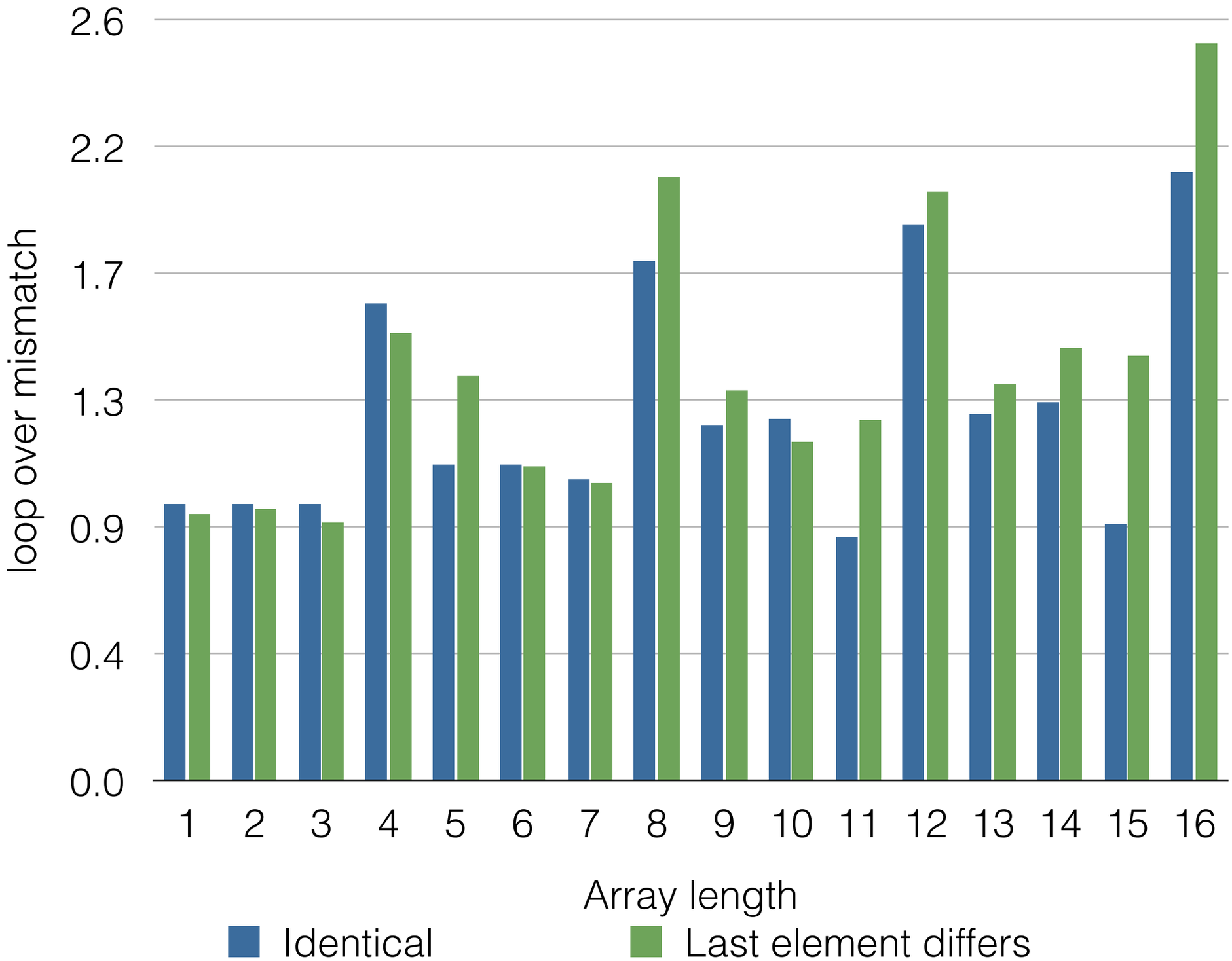
# Internal byte[] mismatch

```
static int mismatch(byte[] a, byte[] b, int length) {
    int i = 0;
    if (length > 3) {
        i = vectorizedMismatch(
                a, Unsafe.ARRAY_BYTE_BASE_OFFSET,
                b, Unsafe.ARRAY_BYTE_BASE_OFFSET,
                length, LOG2_ARRAY_BYTE_INDEX_SCALE);
        if (i >= 0)
            return i;
        i = length + 1 + i; // decode failure index
    }
    for (; i < length; i++) {
        if (a[i] != b[i])
            return i;
    }
    return -1;
}
```

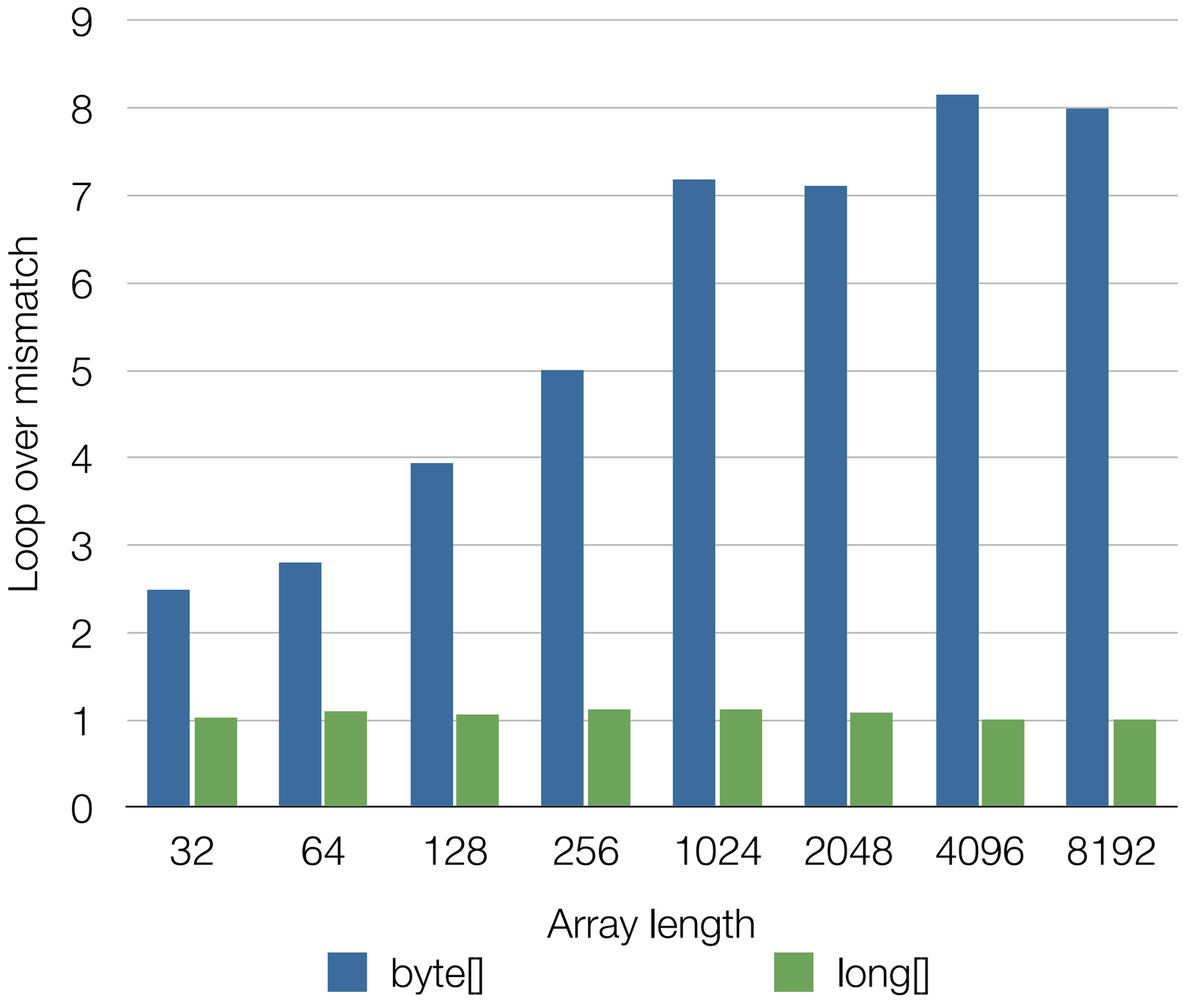
# Internal unsafe mismatch

```
static int vectorizedMismatch(Object a, long aOffset,
                              Object b, long bOffset,
                              int length,
                              int log2ArrayIndexScale) {
    int valuesPerWidth = LOG2_ARRAY_LONG_INDEX_SCALE - log2ArrayIndexScale;
    int wi = 0;
    for (; wi < length >> valuesPerWidth; wi++) {
        long bi = ((long) wi) << LOG2_ARRAY_LONG_INDEX_SCALE;
        long av = U.getLongUnaligned(a, aOffset + bi);
        long bv = U.getLongUnaligned(b, bOffset + bi);
        if (av != bv) {
            long x = av ^ bv;
            int o = BIG_ENDIAN
                ? Long.numberOfLeadingZeros(x) >> (3 + log2ArrayIndexScale)
                : Long.numberOfTrailingZeros(x) >> (3 + log2ArrayIndexScale);
            return (wi << valuesPerWidth) + o;
        }
    }
    ...
}
```

# Small byte[] equals loop over mismatch vs. length



# Large byte[]/long[] equals loop over mismatch vs. length

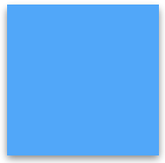


# A good result

- Can get reasonably far with **`sun.misc.Unsafe`**
  - The unsafe aspects are nicely contained
  - HotSpot C2 performance is good for **`byte[]`** and no regression for **`long[]`**
- Reuse **`vectorizedMismatch`** in heap and direct **`Buffer`**s

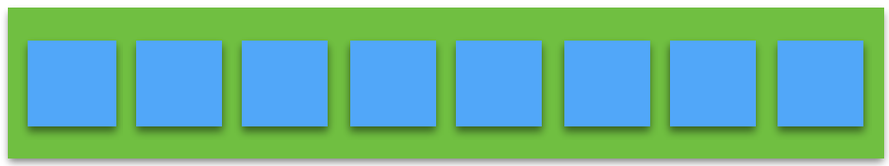
# Can we do better?

**byte 8** bits



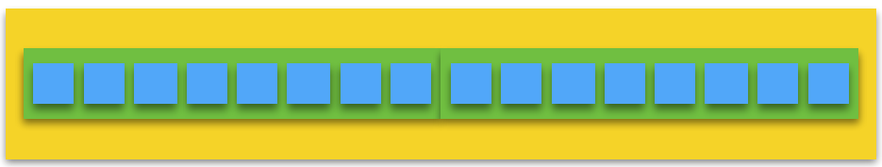
**al 8086**

**long 64** bits



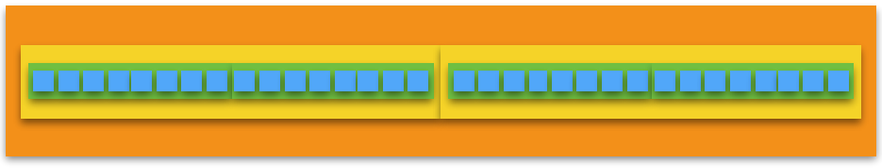
**rax x86**

**Long2 128** bits



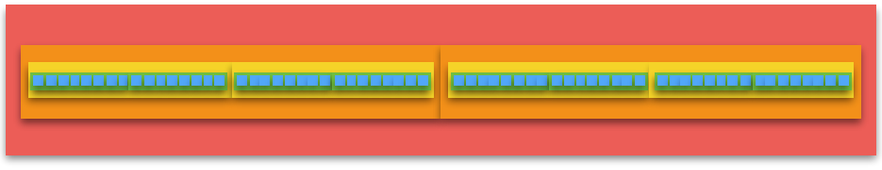
**xmm SSE**

**Long4 256** bits



**ymm AVX**

**Long8 512** bits



**zmm AVX-512**

# Closer to the metal

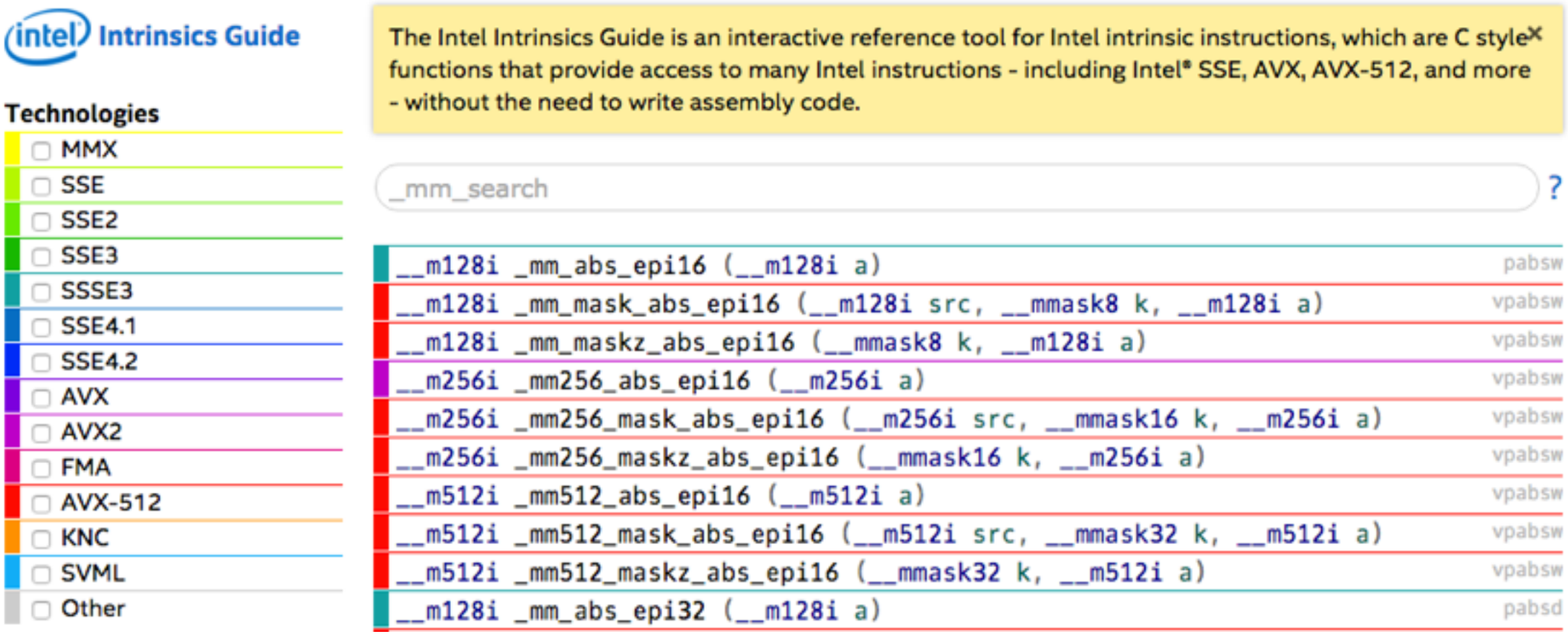
- Make **`vectorizedMismatch`** an intrinsic using SIMD-based machine instructions
  - More reliable than patterns (“lucky loop”)
- Potentially GC specific intrinsics for
  - **`Arrays.equals(char[], char[])`**
  - **`String.equals/compareTo`**

# Is there a way we can experiment without swimming in HotSpot C2's sea of nodes and writing C++?

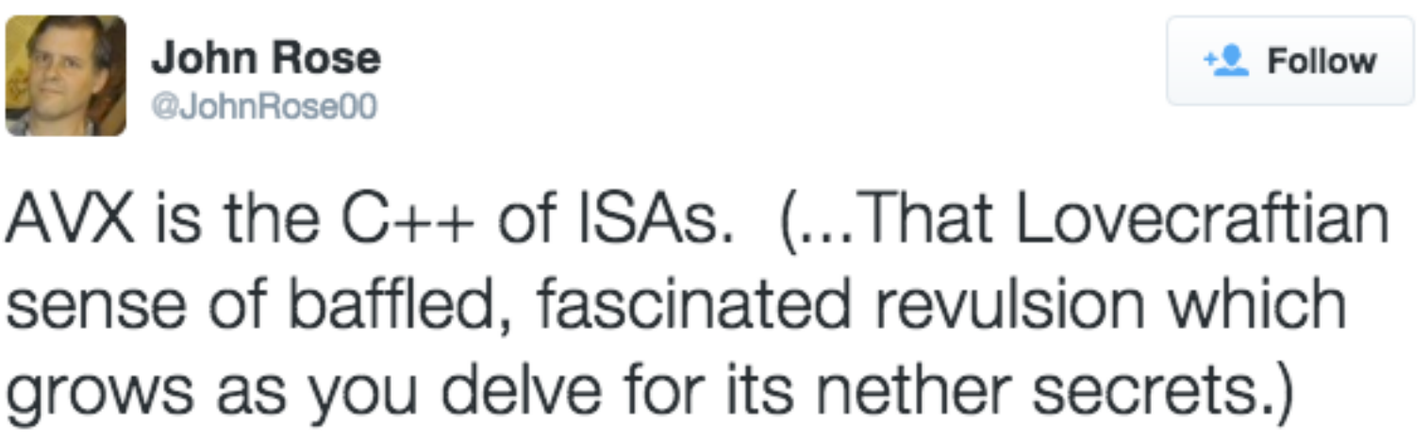
# “Close to the metal” with Project Panama

- Native **`MethodHandle`**s to x86/asm/avx code snippets
  - Calls to snippets can be inlined
- Re-write **`vectorizedMismatch`** using native **`MethodHandle`**s to code snippets
- Using patched Panama repo
  (many thanks to Vladimir Ivanov!)

https://software.intel.com/sites/landingpage/IntrinsicsGuide/




https://twitter.com/JohnRose00/status/656187158630273024



# Equals and mask

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x00 | 0x01 | 0x02 | 0x03 | 0x04 | 0x05 | 0x06 | 0x07 | 0x08 | 0x09 | 0x0A | 0x0B | 0x0C | 0x0D | 0x0E | 0x0F |

**pcmpeqb (_mm_cmpeq_epi8)**

| 0x00 | 0x01 | 0x02 | 0x03 | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**=**

| 0xFF | 0xFF | 0xFF | 0xFF | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**pmovmskb (_mm_movemask_epi8) =**

0b00000000_00001111

**!= 0xFFFF, number of trailing 1s =**

**4, mismatching index = 4**

# XOR and test for zero

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x00 | 0x01 | 0x02 | 0x03 | 0x04 | 0x05 | 0x06 | 0x07 | 0x08 | 0x09 | 0x0A | 0x0B | 0x0C | 0x0D | 0x0E | 0x0F |

**pxor (_mm_xor_si128)**

| 0x00 | 0x01 | 0x02 | 0x03 | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA | 0xAA |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**=**

| 0x00 | 0x00 | 0x00 | 0x00 | 0xAE | 0xAF | 0xAC | 0xAD | 0xA2 | 0xA3 | 0xA0 | 0xA1 | 0xA6 | 0xA7 | 0xA4 | 0xA5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**ptest (_mm_test_all_zeros)**

```
ZF = 0
```

**ZF = 0, number of trailing zeros 0s =**

```
32, mismatching index = 32 >> 3 = 4
```

# Handles to snippets

```
static final
MethodHandle pcmpeqb = CodeSnippet.make("pcmpeqb",
    MethodType.methodType(Long2.class,
                          Long2.class, Long2.class),
    <<pcmpeqb %xmm1, %xmm0>>);
```

```
static final
MethodHandle pmovmskb = CodeSnippet.make("pmovmskb",
    MethodType.methodType(int.class,
                          Long2.class),
    <<pmovmskb %xmm0, %rax>>);
```

# Wrapped handles

```
public static Long2 pcmpeqb(Long2 v1, Long2 v2) {
    try {
        return (Long2) pcmpeqb.invokeExact(new Long2(),
                                           v1, v2);
    } catch (Throwable e) {
        throw new Error(e);
    }
}

public static int pmovmskb(Long2 v1) {
    try {
        return (int) pmovmskb.invokeExact(v1);
    } catch (Throwable e) {
        throw new Error(e);
    }
}
```

# Compare and mismatch

```
public static int compare_pcmpeqb(Object a, long oa,
                                  Object b, long ob) {
    Long2 v1 = UNSAFE.getLong2(a, oa);
    Long2 v2 = UNSAFE.getLong2(b, ob);
    Long2 eq = pcmpeqb(v1, v2);
    return pmovmskb(eq);
}

static int mismatch_pcmpeqb(Object a, long oa,
                            Object b, long ob,
                            int length) {
    for (int wi = 0; wi < length; wi += 16) {
        int r = compare_pcmpeqb(a, oa + wi, b, ob + wi);
        if (r != 0xffff) { return ...; }
    }
    return -1;
}
```

# compare_pcmpeqb

```
0x0000000112496b60: mov    %eax,-0x16000(%rsp)
0x0000000112496b67: push   %rbp
0x0000000112496b68: sub    $0x10,%rsp         ;*invokespecial <init>
                                              ; - java.lang.Long2::<init>@3 (line 19)
                                              ; - sun.misc.Unsafe::getLong2@5 (line 242)
                                              ; - intrinsic.Intrinsics::compare_pcmpeqb@16 (line 32)

0x0000000112496b6c: vmovdqu (%rsi,%rdx,1),%xmm0  ;*invokespecial <init>
                                              ; - intrinsic.VecUtils::makeL2@6 (line 616)
                                              ; - intrinsic.Intrinsics::pcmpeqb@3 (line 180)
                                              ; - intrinsic.Intrinsics::compare_pcmpeqb@25 (line 33)

0x0000000112496b71: vmovdqu (%rcx,%r8,1),%xmm1  ;*synchronization entry
                                              ; - intrinsic.VecUtils::makeL2@-1 (line 616)
                                              ; - intrinsic.Intrinsics::pcmpeqb@3 (line 180)
                                              ; - intrinsic.Intrinsics::compare_pcmpeqb@25 (line 33)

0x0000000112496b77: pcmpeqb %xmm1,%xmm0       ;*invokestatic linkToNative
                                              ; - java.lang.invoke.LambdaForm$NMH/41903949::invokeNative_L3_L@17
                                              ; - java.lang.invoke.LambdaForm$MH/483422889::invokeExact_MT@19
                                              ; - intrinsic.Intrinsics::pcmpeqb@8 (line 180)
                                              ; - intrinsic.Intrinsics::compare_pcmpeqb@25 (line 33)

0x0000000112496b7b: pmovmskb %xmm0,%rax       ;*invokestatic linkToNative
                                              ; - java.lang.invoke.LambdaForm$NMH/488970385::invokeNative_L_I@13
                                              ; - java.lang.invoke.LambdaForm$MH/1277181601::invokeExact_MT@16
                                              ; - intrinsic.Intrinsics::pmovmskb@4 (line 200)
                                              ; - intrinsic.Intrinsics::compare_pcmpeqb@32 (line 34)

0x0000000112496b80: add    $0x10,%rsp
0x0000000112496b84: pop    %rbp
0x0000000112496b85: test   %eax,-0x6100b8b(%rip)        # 0x000000010c396000
                                              ;   {poll_return}
0x0000000112496b8b: retq
```

# mismatch_pcmpeqb

```
0x00000001154d5590: movslq %ebp,%r10          ;*i2l  ; - ArrayMismatchTest::mismatch_pcmpeqb@14 (line 85)

0x00000001154d5593: mov    %r10,%r11
0x00000001154d5596: add    (%rsp),%r11
0x00000001154d559a: vmovdqu (%rbx,%r11,1),%xmm0

0x00000001154d55a0: add    0x8(%rsp),%r10
0x00000001154d55a5: vmovdqu 0x0(%r13,%r10,1),%xmm1

0x00000001154d55ac: pcmpeqb %xmm1,%xmm0       ;*invokestatic linkToNative
                                              ; - java.lang.invoke.LambdaForm$NMH/41903949::invokeNative_L3_L@17
                                              ; - java.lang.invoke.LambdaForm$MH/483422889::invokeExact_MT@19
                                              ; - intrinsic.Intrinsics::pcmpeqb@8 (line 180)
                                              ; - intrinsic.Intrinsics::compare_pcmpeqb@25 (line 33)
                                              ; - ArrayMismatchTest::mismatch_pcmpeqb@23 (line 85)

0x00000001154d55b0: pmovmskb %xmm0,%rax       ;*invokestatic linkToNative
                                              ; - java.lang.invoke.LambdaForm$NMH/488970385::invokeNative_L_I@13
                                              ; - java.lang.invoke.LambdaForm$MH/1277181601::invokeExact_MT@16
                                              ; - intrinsic.Intrinsics::pmovmskb@4 (line 200)
                                              ; - intrinsic.Intrinsics::compare_pcmpeqb@32 (line 34)
                                              ; - ArrayMismatchTest::mismatch_pcmpeqb@23 (line 85)

0x00000001154d55b5: cmp    $0xffff,%eax
0x00000001154d55bb: jne    0x00000001154d55d6  ;*if_icmpeq
                                              ; - ArrayMismatchTest::mismatch_pcmpeqb@32 (line 86)

0x00000001154d55bd: add    $0x10,%ebp         ;*iinc
                                              ; - ArrayMismatchTest::mismatch_pcmpeqb@38 (line 84)

0x00000001154d55c0: cmp    %r14d,%ebp
0x00000001154d55c3: jl     0x00000001154d5590  ;*if_icmpge
                                              ; - ArrayMismatchTest::mismatch_pcmpeqb@7 (line 84)
```

# Can we do better?

# Arrays 2.0 and a Vector API

- Experiments in Project Panama will result in more unsafe stuff
- From which we can build safe abstractions for better array access and vectorization
  - Rough sketch of Vector API by John Rose http://cr.openjdk.java.net/~jrose/arrays/vector/Vector.java

# In summary

- **`sun.misc.Unsafe`** is being frozen in Java 9 but still accessible
  - Cloned internally within the JDK, where internal unsafe features will inevitably increase
- A small set of supported use cases in Java 9
- Significant set of supported use cases anticipated with Projects Valhalla and Panama

?

Hackergarten, Java Hub
Track #2
10am-12pm Wed