MTE4JNI: A Memory Tagging Method to Protect Java Heap Memory from Illicit Native Code Access

Huinan Chen
Wuhan University
Wuhan, China
huinan@whu.edu.cn

Jiang Ma*
Guangdong OPPO Mobile Telecommunications Corp., Ltd.
Dongguan, China
majiang@oppo.com

Chun Jason Xue
MBZUAI
Masdar, United Arab Emirates
jason.xue@mbzuai.ac.ae

Qingan Li*
Wuhan University
Wuhan, China
qingan@whu.edu.cn

Abstract

With the proliferation of mobile devices in daily life, ensuring the security and performance of these devices has become crucial. On Android, the Java Native Interface (JNI) acts as a bridge, allowing native libraries to directly access Java heap memory via raw pointers, bypassing Java's built-in safety checks. While this offers powerful functionality and performance, it also threatens the memory safety of the Java heap. Recently, Memory Tagging Extension (MTE) is introduced into the ARM architectures to enhance memory safety, reducing software vulnerabilities caused by illegal memory operations. This paper proposes MTE4JNI, an MTE-based JNI checking method, to protect Java heap memory from illicit native code access. Experimental results on real Android devices demonstrate that, compared to the currently employed guarded copy method, the proposed MTE4JNI method provides superior memory safety protection, while significantly reducing the runtime overhead on average by 11x and 27x for single-threaded and multi-threaded environments, respectively.

CCS Concepts: • **Security and privacy** → **Mobile platform security**.

Keywords: Memory Tagging Extension, Java Native interface, Memory Security, Java Virtual Machine

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CGO '25, March 01–05, 2025, Las Vegas, NV, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708933

ACM Reference Format:
Huinan Chen, Jiang Ma, Chun Jason Xue, and Qingan Li. 2025. MTE4JNI: A Memory Tagging Method to Protect Java Heap Memory from Illicit Native Code Access. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3696443.3708933

1 Introduction

With the widespread use of mobile devices in our daily lives, the security of mobile applications has become increasingly important. According to information published by Google on its security blog, over 75% of vulnerabilities on the Android platform are due to memory safety violations [22]. To mitigate memory safety violations, most applications on Android devices are written in Java or Kotlin, which run in the Java Virtual Machine (JVM) environment with automatic memory management and various safety checks, including strong typing, array bounds checking, etc.

However, alongside the JVM, developers often utilize native code for the underlying features of the operating system or for runtime performance. As a result, Java applications inevitably need to call native binary code written in native programming languages like C++ via the Java Native Interface (JNI) [12, 16]. During the execution of native code, the safety checks present in the JVM are absent. In JNI, certain functions allow native code to obtain the raw memory address of Java heap objects. With these raw pointers, the native code, outside of the JVM, may manipulate the Java heap memory in an unrestricted manner, such as performing pointer arithmetic. These unsafe manipulations of Java heap memory may lead to undetected memory safety violations such as buffer overflows, which could cause unpredictable harm to the subsequent execution of the program [23–25], affecting the protection of user data and the stability of applications. To detect whether a native method has performed illegal memory operations, such as out-of-bounds access using pointers returned by the JNI interface, Android Runtime (ART) currently uses the *guarded copy* method[2], which supports limited out-of-bound memory access checking, as

discussed in Section 2.3, at the cost of significant runtime overhead to slowdown the JNI interface performance by 20X, as illustrated in Section 5.3, which hinders guarded copy from the default production configuration.

ARM introduced the Memory Tagging Extension (MTE) [3] recently in its ARMv8.5-A architecture, to enhance memory safety by helping to detect memory corruption issues such as buffer overflows and use-after-free errors. MTE helps identify memory access violations by tagging memory regions and checking these tags during access. There are many recent studies aimed at enhancing program security[13, 19, 26] with this MTE feature.

The support for MTE was introduced in Android 12 to enhance the memory safety and security of native memory. If ARM's MTE is integrated into the Android Runtime (ART) JNI interface, it could provide developers with a secure runtime environment to detect vulnerabilities during the development phase. This integration would not only help prevent attacks exploiting memory safety issues in Java native methods but also significantly enhance the overall security of Android applications. However, to the best of our knowledge, no prior work has been proposed to utilize the MTE feature specifically for safeguarding Java heap memory, despite its critical role in protecting user data and ensuring stable program execution.

To fill this void, this paper proposes MTE4JNI, an MTE-based JNI checking method, to protect Java heap memory from illicit native code access. The method leverages MTE features to set memory tags on Java objects accessed by native methods, and pointer tags on pointers returned to native methods. This allows the MTE mechanism to detect the out-of-bound memory accesses by comparing each memory tag and the corresponding pointer tag. Experimental results demonstrate that, compared to the guarded copy method supported in the Android runtime, the proposed MTE4JNI approach provides enhanced memory safety protection, with tolerable runtime overhead, demonstrating the potential for MTE4JNI to be applied in production environments. The contributions of this work are summarized as follows:

- We propose a novel memory safety protection method, MTE4JNI, to protect Java heap memory from illicit native code access;
- Based on ARM's Memory Tagging Extension (MTE), we implement the MTE4JNI approach into the Android runtime;
- We conduct a comprehensive set of experiments to evaluate the performance and security enhancement of our proposed method. Experimental results on real Android devices demonstrate that, compared to the currently employed guarded copy method, the proposed MTE4JNI method provides superior memory safety protection, while significantly reducing the runtime overhead on average by 11x and 27x for single-threaded and multi-threaded environments, respectively.

2 Background and Motivation

2.1 Memory Tagging Extension

ARM's Memory Tagging Extension (MTE) is an innovative hardware security feature introduced in the ARMv8.5a instruction set, designed to help detect and prevent memory-related vulnerabilities, such as buffer overflows and use-after-free errors [3].

The basic idea of applying MTE mechanism for memory safety works is as below. Upon the creation of a memory block, i.e., at allocation time, MTE assigns the same tag to a memory block (called *memory tag*) and to the returned memory pointer (called *pointer tag*). Ideally, it is expected that the memory tags can be used to differentiate different allocations of memory blocks. Later, on each memory access with this pointer, the memory tag and the pointer tag are checked and a mismatch indicates an unsafe memory access.

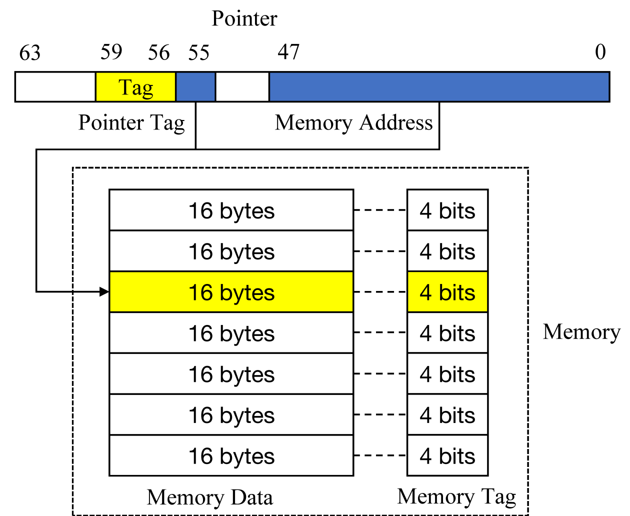

Figure 1. Overview of Memory Tagging Extension

Specifically, according to the ARM MTE specification, each 16-byte aligned memory unit shares a single memory tag. The memory tags are stored in system memory and can be cached by the CPU. As shown in Figure 1, the pointer tag is placed in the 56-59th bits of the returned pointer, providing 4 bits in total. Note that a 64-bit address used only the lowest 48 bits, while the 48-64th bits are reserved. When accessing a memory block via a pointer, the processor compares the pointer tag stored in the 56-59th bits of this pointer, with the corresponding memory tag of this pointer (memory address). ARMv8.5a provides specific instructions to support this work.

For example, in the case of out-of-bound memory access, the memory tag tag_o of the out-of-bound memory block will be different from that tag_i of the in-bound memory block, with the rare exception that these two blocks belong to the same aligned 16-byte memory block. As stated above, since the pointer tag is stored in the reserved bits of the pointer itself, the new pointer generated by pointer arithmetic will inherit these bits and thus keep the in-bound pointer tag tag_i unchanged. As a result, upon an out-of-bound memory access, a mismatch between the memory tag tag_o and the pointer tag tag_i will be detected.

The Linux kernel currently supports ARM's Memory Tagging Extension (MTE), offering various error-checking modes to determine the most suitable memory safety mechanism, for different requirements of performance and security in various applications[1]. The error-checking modes provided by Linux mainly include *synchronous* and *asynchronous* modes. The synchronous mode checks the tag consistency immediately upon memory access, and if a mismatch is detected, an exception is synchronously generated, allowing developers to promptly locate and diagnose memory-related issues. The asynchronous mode allows the program to continue execution even after detecting a tag mismatch, only logging the error occurrence. This mode is suitable for scenarios with high-performance requirements where the overhead of synchronous checking cannot be tolerated.

2.2 Native Method and Java Native Interface

In Java, a *native method* is a method that is implemented in platform-specific code, typically in C or C++. It allows Java applications to call native libraries that is not written in Java, such as system APIs or high-performance mathematical libraries. This help Java applications to perform tasks that are challenging to execute within the Java environment, such as direct access to operating system features or high-efficiency computations.

The Java Native Interface (JNI) is a standard programming interface on the Java platform that allows Java code to interact with native code written in other languages. JNI serves as a bridge between Java and native methods, as well as between native code and Java methods. Through JNI, not only Java code can invoke native code, but also native code can create new Java objects, invoke methods of Java classes, read and write fields of Java objects, catch and throw exceptions, and more.

One of the common tasks in native methods is to manipulate objects in the Java heap. In JNI, some interfaces may return a raw pointer of the Java heap object to native methods, as illustrated in the first column of Table 1. If the raw pointers returned by these JNI interfaces are used carelessly or maliciously by the native code, it may corrupt the Java heap memory while bypassing the memory safety checks enforced by the JVM. Therefore, buggy native code can compromise the memory safety of the Java heap, leading to security vulnerabilities or program crashes. After the native code finishes its work with this pointer, the corresponding release interfaces are used to release these pointers, as illustrated in the second column of Table 1.

2.3 Guarded Copy

To detect whether a native method has performed illegal memory operations, such as out-of-bounds accesses using pointers returned by the JNI interface, the ART currently uses the *guarded copy* method[2]. As shown in Figure 2, the basic idea of guarded copy is that: (1) When native code requests the address of a heap object, the object is copied, and two red zones, prefilled with a specific repeating canary pattern string, are added before and after the copy respectively; (2) After the native code has finished manipulating the object, the red zones are checked to ensure each value still matches the canary string before releasing. This checks whether out-of-bound writes occur during the execution of the native code; (3) If changed, an out-of-bounds memory operation is detected. Otherwise, the copy is used to update the original heap object.

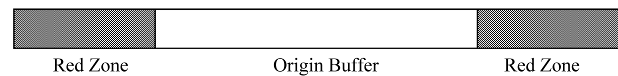

Figure 2. The guarded copy method

However, this method has several limitations. Firstly, it can only detect out-of-bounds write accesses, not out-of-bounds reads, since out-of-bounds reads never change values. Secondly, if the out-of-bounds access surpasses and skips the red zones, the error cannot be detected. Thirdly, this method has a significant impact on performance due to the introduced memory copying and synchronization, as discussed in Section 5.3. Lastly, it can only detect whether an error has occurred, without providing more detailed information.

2.4 Motivation and Challenges

This paper is motivated to propose a memory tagging method to protect Java heap memory from illicit native code access. We can modify the JNI interfaces in Table 1 to specifically allocate the memory tag for the memory block of the array or string before returning the pointer. The pointer would then be returned to native code with the corresponding pointer tag. Enabling the MTE-based error-checking mode, native code can access the array normally, while any attempt to perform out-of-bounds access using the pointer would be immediately detected by the processor, triggering an exception. After the native code finishes its work with this pointer, we can modify the JNI release interfaces in Table 1 to release the memory tag for the corresponding memory block.

Table 1. JNI Interfaces Returning Raw Pointers to Heap Memory

Get interface	Release interface	Pointers to
GetStringCritical	ReleaseStringCritical	String
GetPrimitiveArrayCritical	ReleasePrimitiveArrayCritical	Primitive array
GetStringChars	ReleaseStringChars	String
GetStringUTFChars	ReleaseStringUTFChars	UTF-encoded String
Get*ArrayElements	Release*ArrayElements	Primitive array
Get*ArrayRegion	Release*ArrayRegion	Portion of primitive array

* can be any of the following primitive types: byte, char, short, int, long, float, double

However, there are a couple of challenges to overcome to enable this MTE-based JNI checking in Android systems for multi-threaded scenarios.

- A unique memory tag should be allocated to an object's memory block upon returning its pointer, and applied to both the pointer and the memory block. However, Android applications are commonly highly multi-threaded, and native code threads may concurrently obtain the raw pointer of the same Java object. In such cases, it is critical to efficiently share the same tag among concurrent threads and ensure that the tag is only released after all threads have finished using it.
- When a JNI native thread accesses a tagged memory block, there may be concurrent supporting threads, such as the garbage collection thread, accessing the same memory block with non-tagged pointers. This could lead to segmentation faults due to failed tag checking. Differentiating between these concurrent threads and enabling MTE checking mechanisms at the thread level is crucial. This challenge is further exacerbated by Android's complicated architecture, making it necessary to integrate MTE functionality without disrupting system compatibility.

3 The Design of MTE4JNI

This paper proposes a memory tagging method called MTE4JNI, to protect Java heap memory from illicit native code access. The proposed MTE4JNI method includes the following three parts:

- Memory tag allocation. Before a specific JNI interface returns the raw pointer of a Java object to the native code, an appropriate tag should be allocated to the 56-59th bits of the pointer, and the same tag should also be allocated to the addressed memory block, which may consist of multiple 16-byte aligned memory sub-blocks.
- Memory tag release. After the native code has finished using the pointer and invoked the JNI release interface to release it, the memory tag of the object must be released in time to reduce the portability of tag conflict. Otherwise, different allocations of memory blocks sharing the same tag may confuse the error-checking of MTE.
- MTE enabling. The MTE error-checking feature should be enabled or disabled at appropriate moments to avoid errors during concurrent access to heap objects by other threads such as garbage collection threads.

3.1 Memory Tag Allocation

Memory tag allocation is required before a specific JNI interface returns the raw pointer of a Java object to the native code. This work aims to address the challenge of concurrent memory access to a Java object in a multi-threaded environment. A unique memory tag should be allocated to an object's memory block upon returning its pointer to the native code, to apply the same tag to the pointer and the memory block. However, in a multi-threaded environment, threads of native code may obtain the raw pointers of the same Java object concurrently. Hence it is critical to share the same tag for all the returned pointers as well as the object in an efficient way.

A naive solution is to provide a global lock for the memory tag allocation work. Each of concurrent native thread should acquire the lock first and then the memory tag allocation can be conducted exclusively. However, this naive solution is too coarse-grained and will cause significant overhead, as every thread of JNI interfaces must compete for this global lock. To address this issue, we design a memory tag allocation algorithm based on reference counting with a two-tier locking scheme.

3.1.1 The Reference Counting Scheme.

Before a memory tag is allocated in a thread of native code, we use a reference count to track whether other threads are already executing native code and have obtained a pointer to a Java object. If the reference count indicates that the object has already been tagged, there is no need to generate a new tag. We simply increment the reference count by one and directly return the tagged pointer. This enables concurrent threads to share the same tag. Otherwise, if there is no other thread of native code holding this object, we generate a random tag, setting it to both the memory tag as well as the pointer tag. Then, the reference count increments by one, and the tagged pointer is returned.

3.1.2 The Two-tier Locking Scheme.

Since we have introduced a global reference count for each object accessed by native code, there is a risk of concurrent access to the reference count itself. A naive idea is to introduce a global lock to ensure that only one thread can access or modify the reference count at a time. However, using a global lock to ensure thread safety may severely impact performance. Therefore, we propose a finer-grained two-tier locking scheme to address this issue.

We maintain k hash tables. In a hash table, each key represents the starting address of an object, and each value is a tuple with two elements, where one element is the address of the reference count for the object, and the other is the address of the dedicated lock for the object. The set of Java objects simultaneously accessed by native code are distributed across k hash tables based on the lower part of each object's starting address.

Each hash table is guarded with a dedicated *table lock* to ensure thread safety. When a thread of native code accesses a hash table to obtain the reference count address and the dedicated lock address of an object, the corresponding table lock must be held. Once the addresses are retrieved, the table lock is immediately released. Similarly, each object is guarded with a dedicated *object lock* to ensure thread safety. When a thread of native code accesses the reference count and memory tag of an object, the object lock must be held. Once this task is finished, the object lock is released immediately.

3.1.3 The Complete Memory Tag Allocation Algorithm.

Combining the aforementioned reference counting mechanism and the two-tier lock mechanism, the memory tag allocation algorithm works as follows. Firstly, determines the target hash table by directly using the lowest part of the object's address as the index. Secondly, retrieve or create the reference count for this object. Thirdly, retrieve or create the memory tag for this object. Lastly, generate the tagged pointer and return. The detailed algorithm is illustrated in Algorithm 1. This algorithm resolves the issue of concurrent access in a multi-threaded environment while minimizing the performance impact caused by locking.

Algorithm 1: The Memory Tag Allocation Algorithm

```
Input  : Memory start address begin, memory end
         address end
Output : Tagged pointer to the memory region

// 1. Determine which hash table to use based on begin address
hashTableIndex ← begin/16  mod 16;
// 2. Retrieve or create the reference count
lock the specific hash table's lock
  hashTableLocks[hashTableIndex];
if begin exists in the hash table
  hashTables[hashTableIndex] then
    Retrieve referenceNum and mutexAddr from the
      hash table;
else
    Create new referenceNum and mutexAddr;
    Insert begin → {referenceNum, mutexAddr} into
      hashTables[hashTableIndex];
unlock the specific hash table's lock
  hashTableLocks[hashTableIndex];
// 3. Retrieve or create the memory tag
lock mutexAddr;
Increment referenceNum;
if referenceNum > 1 then
    Load existing memory tags using ldg instruction;
else
    Generate new memory tags using irg instruction;
    Apply new tags to memory from begin to end
      using st2g and stg instructions;
// 4. Generate tagged pointer and return
unlock mutexAddr;
Generate tagged pointer by applying the memory tag
  to begin address;
return Tagged pointer;
```

3.2 Memory Tag Release

Memory tag release is required after the native code has finished using the pointer and invoked the JNI interface to release it. It decrements the reference count by one first. When the updated reference count becomes zero, it indicates that no other thread of native code is holding a pointer to this object, allowing us to safely release the memory tags for this object.

This work also needs to ensure thread safety in a multi-threaded environment. We rely on the two-tier locking scheme described earlier to guard the access to each hash table by a table lock and guard the access to the reference count and memory tag of an object by an object lock.

The memory tag release algorithm works as follows. Firstly, determines the target hash table by directly using the lowest part of the object's address as the index. Secondly, retrieve the reference count for this object. Finally, optionally releasing the memory tag. The detailed algorithm is illustrated in Algorithm 2.

3.3 Enabling the MTE Error Checking Mechanism

Currently, the Linux system provides some support for ARM MTE, but we can only use the *prctl* system call to enable or disable the MTE error-checking mechanism at the process level. However, during the actual execution of a Java application, when one user thread enters the native code section to access the Java memory, other supporting threads such as the

Algorithm 2: The Memory Tag Releasing Algorithm

```
Input  : Memory start address begin, memory end
         address end
Output : None

// 1. Determine which hash table to use based on begin address
hashTableIndex ← begin/16 mod 16;
// 2. Retrieve the reference count
lock the specific hash table's lock
  hashTableLocks[hashTableIndex];
if begin exists in the hash table
  hashTables[hashTableIndex] then
    Retrieve referenceNum and mutexAddr from the
      hash table;
else
    // If no entry exists, nothing needs to be done
    unlock the specific hash table's lock
      hashTableLocks[hashTableIndex];
    return;
unlock the specific hash table's lock
  hashTableLocks[hashTableIndex];
// 3. Optionally releasing the memory tag
lock mutexAddr;
Decrement referenceNum;
if referenceNum == 0 then
    // Release memory tags if the reference count is zero
    Set the memory tags to 0 from begin to end;
unlock mutexAddr;
```

garbage collection (GC) thread, may also be running simultaneously. For example, the GC thread could potentially access the same tagged Java memory, but the pointer in the GC thread never walks through the JNI interface to be tagged. It leads to a mismatch and thus an exception of memory access violation, regardless of in-bounds or out-of-bounds memory accesses.

To address this issue, we need a thread-level control to enable MTE error-checking mechanism for user threads with native code. Although Linux does not provide such an interface, according to ARM's documentation, we can control MTE at the thread level by setting the TCO system register.

Specifically, in Android when a Java thread invokes native code, a trampoline function is used to handle tasks such as Java thread state transitions and parameter conversions. We can modify the trampoline function to include instructions that modify the TCO register to enable MTE. Similarly, after the native code execution is completed, another trampoline function is used to return the thread control flow back to the Java code. We can modify this trampoline function to modify the TCO register to disable the MTE error-checking mechanism. It enables the MTE error-checking mechanism only for threads with native code, ensuring that no exceptions are triggered due to a mismatch between the pointer tag and memory tag when threads are not executing native code.

4 Implementation Analysis

The proposed approach is implemented in the Android Runtime, with modifications to the following three parts.

- Heap Memory Allocation: The heap memory allocation work is modified to adjust the default alignment of memory allocation to be consistent with the MTE mechanism, and also enable the MTE checking for the allocated memory.
- JNI Interfaces: The code within JNI interfaces is modified to integrate the MTE4JNI mechanism, involving adding the code for memory tag allocation and release, respectively.
- Trampoline Functions: The trampoline functions are modified to flexibly enable or disable the MTE error-checking mechanism at the thread level.

4.1 Heap Memory Allocation

As discussed above, in the MTE mechanism, memory tags are set at a 16-byte granularity. The ART's memory allocator has the default alignment of 8 bytes, which means the starting memory address as well as the size of each allocated object is a multiple of 8. This mismatch between the alignment of object allocation and the granularity of memory tagging brings up an issue. Two objects may be allocated into a single memory block of 16-byte, and thus sharing the same tag. As a result, the MTE error-checking mechanism is confused to view the out-of-bounds access within the same block as a safe one.

To address this issue, we need to modify the alignment in ART's memory allocation from 8 bytes to 16 bytes. Although 16-byte alignment may cause minor internal memory fragmentation, this overhead is generally negligible given that Java objects are relatively large. Moreover, many alignment methods on 64-bit machines default to 16 bytes. During the heap memory allocation phase, we also need to modify the protection flags of the heap's memory-mapped region by adding *PROT_MTE* to indicate that the memory will use memory tagging for enhanced safety and security.

4.2 JNI Interfaces

As discussed above, the memory tag allocation work should be conducted in each JNI interface that can potentially obtain a raw pointer to the Java heap objects, to check the safety of memory accesses due to this pointer. As a result, no matter whether an object is of a string or an array of any primitive type, as long as its heap memory address is returned directly, it will always undergo memory tag allocation before the native code obtains its memory address. Similarly, the memory

tag release work should be conducted in the corresponding JNI release interface to recycle the memory tags.

4.3 Native Method Trampoline Function

As discussed above, the MTE checking mechanism can be enabled or disabled at the thread level by setting the TCO system register. Regarding the placement of the enabling or disabling code, there are three types of native methods to consider.

In Android, native methods can be annotated with *@FastNative* and *@CriticalNative*. For native methods annotated with *@CriticalNative*, since they are optimized for maximum performance and never have access to Java heap objects, we do not need to consider them. For native methods annotated with *@FastNative*, since they do not involve thread state transitions, we need to set the TCO register directly in both the specifically compiled trampoline functions and the generic trampoline functions. For regular native methods annotated with neither *@FastNative* nor *@CriticalNative*, the trampoline function always invokes the Java thread state transition function. Therefore, we insert the code to set the TCO register within this transition function.

5 Evaluation

5.1 Experimental Setup

The proposed MTE4JNI method is implemented in an OPPO Find N2 Flip mobile phone with the Color OS 14.0 based on Android 14, as illustrated in Table 2. To comprehensively evaluate the proposed MTE4JNI method compared to the existing guarded copy method in Android systems, we conducts the following experiments. In this evaluation, we use 16 hash tables in the MTE4JNI method.

Table 2. Experimental Environment Configuration

Parameter	Configuration
Experimental Device	OPPO Find N2 Flip
SoC	MediaTek Dimensity 9000+
RAM	12GB
System Environment	Color OS 14.0 based on Android 14

- The evaluation of the effectiveness of error checking. This experiment is to verify that the proposed MTE4JNI method can accurately identify illegal memory operations in native code;
- The evaluation of runtime overhead for the JNI interfaces. This experiment is to verify that the MTE4JNI method incurs limited performance overhead to the JNI interfaces;
- The evaluation of runtime performance for other common tasks. This experiment aims to verify that the MTE4JNI method incurs a limited performance overhead for other common tasks.

We compared four schemes as follows:

- No protection scheme: by default, the JNI out-of-bounds checking is disabled for performance reasons;
- Guarded copy scheme: the existing guarded copy-based JNI out-of-bounds checking is enabled with specific options;
- MTE4JNI+Sync scheme: the proposed MTE4JNI method is enabled in the synchronous mode;
- MTE4JNI + Async scheme: the proposed MTE4JNI method is enabled in the asynchronous mode.

5.2 Effectiveness of Out-of-bounds Checking

We run a test program that triggers a buffer overflow to detect how various methods identify the buffer overflow error. In the test program, the Java program invokes a native method, and the native method uses the *GetPrimitiveArrayCritical* interface to obtain a pointer to the array object in the Java heap and performs an out-of-bounds access, incorrectly modifying the memory location outside the array boundary. The core native code is illustrated in Figure 3.

Under the no protection scheme, this out-of-bounds write cannot be checked, and the program terminates normally, unaware of the unsafe memory write. Under the guarded copy scheme, the ART detects the out-of-bounds modification to the array and provides the offset of the incorrect modification by the native method. However, the error is detected in the JNI release interface, instead of the real native code doing the out-of-bounds access. The stack trace information recorded is far away from the faulting location, as illustrated in Figure 4a.

```
jboolean test_ofb(JNIEnv *env, jobject thiz,
                  jintArray array1) {
    jboolean is_copy1;
    jint *elems1 = (jint *) env ->
        GetPrimitiveArrayCritical(array1, &is_copy1);
    elems1[21] = 50;
    env->ReleasePrimitiveArrayCritical(array1, elems1, 0);
    return JNI_TRUE;
}
```

Figure 3. The core code of the native method. The original Java object is an array of 18 integers, but the native code writes into the array with the index of 21, leading an out-of-bounds write.

Under the synchronous error-checking mode of MTE, the out-of-bounds access is detected and a segmentation fault is reported immediately after the native method performs an out-of-bounds access using the pointer. The stack trace information recorded is accurate enough to allow precise pinpointing of the faulting location, as illustrated in Figure 4b.

Under the asynchronous error-checking mode of MTE, the out-of-bounds access is detected and a segmentation fault is reported in a delayed fashion. The stack trace recorded

is the first system call or context switch that was executed after the memory corruption happened, which is far away from the faulting location, as illustrated in Figure 4c.

5.3 Runtime Overhead of JNI Interfaces

The JNI checking methods, either the guarded copy method or the proposed MTE4JNI method, introduce additional computation tasks to the JNI interface, inevitably causing runtime overhead. We evaluated the runtime overhead of the JNI interfaces in both single-thread and multi-thread environments.

5.3.1 Single-thread Performance. To evaluate the runtime overhead in a single-thread environment, we executed a native method that initially obtains pointers to two Java integer array objects via a JNI interface. The method then copies the contents of one array into the other and finally releases the pointers using the corresponding JNI release interface.

For a comprehensive evaluation, we provided multiple versions of array objects with various lengths ranging from 2^1 to 2^{12} elements of integer, as most objects in popular Android applications are smaller than 2^{14} bytes [8]. The execution time of this native method was recorded. Figure 5 shows the execution time for copying arrays of different lengths under various schemes, normalized to the execution time without protection. Three observations can be made.

The first observation is that for various array lengths, the guarded copy scheme always causes the largest overhead, which is far more than the other schemes. On average, the guarded copy scheme slowdowns the native method by **26.58** times, while the MTE4JNI-Sync and MTE4JNI-Async schemes slowdowns the native method by only **2.36** times and **2.24** times, respectively.

The reason is that different JNI checking schemes introduce additional computation tasks into the JNI interfaces in different ways. The guarded copy method allocates a memory block, copies the original Java object to the new memory block with two red zones on both sides, and performs operations on this copy. In comparison, the MTE4JNI schemes allocate a tag for the memory block as well as the returned pointer and operate directly on the original object. As a result, the MTE4JNI schemes cause much less overhead than the guarded copy scheme.

The second observation is that the MTE4JNI-Async scheme is slightly faster than the MTE4JNI-Sync scheme. This is because, in the asynchronous mode, tag checks are deferred, allowing for more efficient processing without blocking.

The third observation is that under all JNI checking schemes, as the array size increases, the slowdown becomes smaller. This is because the workload of copying the array increases linearly with the array size, while the introduced overhead increases sublinearly, leading to reduced relative overhead for larger arrays.

5.3.2 Multi-thread Performance. Because the MTE4JNI method involves the use of locks, there will inevitably be some negative impact on performance in multi-thread scenarios. This evaluation was divided into two tests. In the first test, multiple threads concurrently read on the same array. In the second test, each of the concurrent threads reads its own specific array. Concurrent threads on the same array must compete for the exclusive locks associated with each object, *object lock*, while concurrent threads on different arrays must compete for the hash table locks, *table lock*. These tests were conducted to thoroughly investigate the impact of our locking mechanism on multi-threaded performance. For each test, we compared three methods, the proposed MTE4JNI method with the two-tier locking scheme, MTE4JNI with a naive global lock, and the guarded copy method. The first two methods were evaluated in synchronous and asynchronous modes separately.

We created 64 threads to concurrently run a native method that repeatedly reads an array containing 1024 integers 10000 times. In the first test, each thread attempts to access the same array. In the second test, each thread attempts to access its own specific array. We then recorded the time taken for all threads to complete their tasks, as illustrated in Figure 6.

Firstly, it is found that in the first test of assessing the same array, the execution times of the two-tier locking, global locking and guarded copy schemes are **1.21x**, **1.39x**, and **32.9x**, respectively. In the second test, the execution times are **1.21x**, **2.20x**, and **34.0x**, respectively. This is because, compared to the global locking scheme, the two-tier locking scheme shortens the critical sections to mitigate stalls due to lock contention, thus leading to the best performance. The guarded copy method involves much effort for object copying, thus leading to the worst performance.

Secondly, it is found that compared with the first test, the performance gap between the proposed two-tier locking scheme and the other two schemes is significantly widened in the second test. This is because when accessing different arrays, in the two-tier locking scheme threads never compete for the object lock, and compete for the table lock only when the addresses of these two arrays coexist in the same hash table, which significantly mitigates the lock contention. However, in the same situation, the global locking scheme still causes lock contention for every thread of JNI interface, and the guarded copy scheme involves copying more objects.

5.4 Common Task Performance Test

To assess the impact of our MTE4JNI scheme on the performance of commonly seen tasks, we conducted performance experiments using the CPU test suite of GeekBench 6.3.0, a widely used performance testing tool on Android devices [5]. It covers a range of workloads designed to evaluate and optimize CPU and memory performance. These workloads

```
40 total frames
backtrace:
    #00 pc 000000000005e084  /apex/com.android.runtime/lib64/bionic/libc.so (abort+180) (
        ↪ BuildId: d0191b43bb3a9e52db42d88a17e7d136)
    #01 pc 000000000064bfd0  /apex/com.android.art/lib64/libart.so (art::Runtime::Abort(char
        ↪  const*)+1536) (BuildId: 00404bfd7d12f9b389b42cb48f708929)
```

(a) The stack trace information from logcat under the guarded copy scheme. ART detects an error on releasing the pointer and invokes *abort*, resulting in the top stack trace showing an *abort* function.

```
29 total frames
backtrace:
    #00 pc 000000000001fd88  /data/app/~~d9mEYJ_Ny9pfzvHz1OL5WA==/com.example.
        ↪ mtetestoutofbounds-24lFxXaf_KSukqt8du-TOw==/base.apk!libmtetestoutofbounds.so (
        ↪ offset 0x80000) (test_ofb(_JNIEnv*, _jobject*, _jintArray*)+124) (BuildId: 44
        ↪ f82c0a69c825b7baddbd8d61ade517e7102d70)
    #01 pc 0000000000020064  /data/app/~~d9mEYJ_Ny9pfzvHz1OL5WA==/com.example.
        ↪ mtetestoutofbounds-24lFxXaf_KSukqt8du-TOw==/base.apk!libmtetestoutofbounds.so (
        ↪ offset 0x80000) (Java_com_example_mtetestoutofbounds_MainActivity_mteTest
        ↪ GetPrimitiveArray+40) (BuildId: 44f82c0a69c825b7baddbd8d61ade517e7102d70)
```

(b) The stack trace information from logcat under the MTE4JNI-Sync scheme. An out-of-bound access triggers an immediate error detection, showing the exact faulting instruction in the native method at the top of the stack trace.

```
39 total frames
backtrace:
    #00 pc 00000000000b65c4  /apex/com.android.runtime/lib64/bionic/libc.so (getuid+4) (
        ↪ BuildId: d0191b43bb3a9e52db42d88a17e7d136)
    #01 pc 000000000000d824  /system/lib64/liblog.so (LogdWrite(log_id, timespec*, iovec*,
        ↪ unsigned long)+180) (BuildId: e7bafa38098207842c4cb0a44eeb8e25)
```

(c) The stack trace information from logcat under the MTE4JNI-Async scheme. The error detection is deferred until the next syscall (getuid), as illustrated at the top of the stack trace.

Figure 4. Stack trace information from logcat under different schemes.

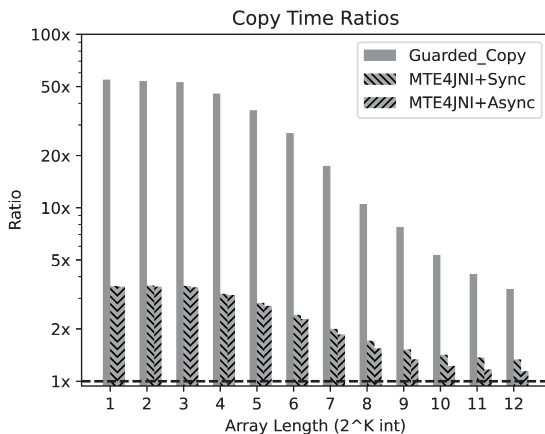

Figure 5. The execution time of the native method for copying one array into another in a single-threaded environment, normalized to the no-protection scheme. The y-axis shows logarithmic values for better comparison.

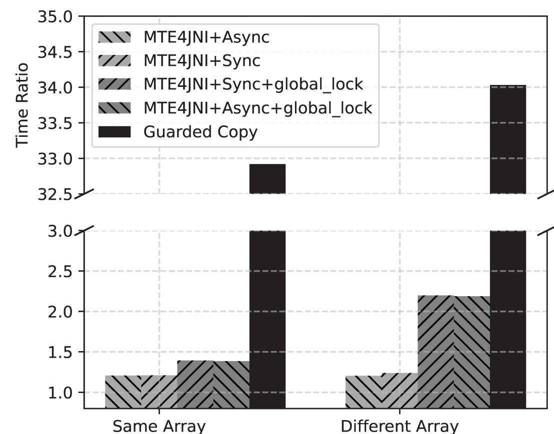

Figure 6. The execution time for concurrently reading an array in a multi-threaded environment, normalized to the execution time under no protection.

include data compression, image processing, machine learning, and code compilation. These workloads are critical for various applications, including web browsers, image editors, and development tools.

The experiment evaluated GeekBench 6.3.0 in Android AArch64 under four schemes: the no protection scheme, the guarded copy scheme, the MTE4JNI+Sync scheme, and the MTE4JNI+Async scheme.

Figures 7 and 8 show the single-core and multi-core performance scores of the sub-items relative to the no protection scheme, respectively. In the single-core environment, the guarded copy, MTE+Sync and MTE+Async schemes cause performance degradation of 5.90%, 5.33%, and 1.13%, respectively. In the multi-core environment, the guarded copy, MTE+Sync and MTE+Async schemes cause performance degradation of 13.50%, 5.12%, and 1.55%, respectively. Exceptions are the workloads of Clang, Text Processing, and PDF Render, which work worse under the MTE+Sync scheme than under the guarded copy scheme. This reason is, that these workloads typically cause intensive access within a large array. Under the MTE+Sync scheme, each access incurs checking overhead, while under the guarded copy scheme, the whole array incurs the overhead of data copying only once. This provides an insight that such memory-intensive workloads are not suitable for the MTE+Sync scheme.

The results show that compared to the guarded copy scheme, the proposed MTE4JNI scheme demonstrates performance improvements in daily tasks with minimal impact on performance. In a multi-thread environment, the proposed MTE4JNI scheme, combined with the asynchronous error-checking mode, showed a 14% improvement over ART's current guarded copy scheme. These results effectively indicate that the MTE4JNI scheme has a minimal impact on common tasks, proving its feasibility and effectiveness in practical deployment.

6 Related Work

The related work can be divided into hardware-assisted approaches and software-based approaches. The former solutions leverage certain hardware features to enhance security, while the latter can be further divided into native programming language based approaches and JVM runtime based approaches.

6.1 Hardware-assisted Approaches

At the hardware level, ARM introduced Pointer Authentication (PA) in ARMv8.3a to enhance memory safety. PA is designed to defend against pointer-related attacks, such as return address tampering or function pointer hijacking. Subsequently, numerous studies have explored ways to improve software security using PA. The PACMem system is proposed to use ARM's pointer authentication to enhance spatial and temporal memory safety, reducing security threats from memory vulnerabilities with minimal performance impact [11]. PAL is a kernel-level hardware control flow integrity (CFI) protection mechanism using pointer authentication [27]. An improved CFI scheme for embedded devices is proposed to enhance defenses against indirect call attacks with minimal performance overhead [15]. Work explores bypassing ARM's pointer authentication using speculative execution, revealing a new security flaw and presenting mitigation strategies [17]. A toolset leveraging pointer authentication is proposed to protect C and C++ programs against runtime attacks, significantly enhancing security with acceptable performance loss [14].

Following this, ARM introduced the Memory Tagging Extension (MTE) in ARMv8.5a [3]. MTE assigns tags to memory allocations and verifies them during memory access to detect and prevent illegal memory operations, such as memory corruption and out-of-bounds access.

Recently, several new studies have emerged in the field of memory safety based on MTE. A deterministic memory protection design based on MTE is integrated into LLVM Clang, enhancing memory safety with minimal runtime and code size overhead [13]. Multi-Tag, a design employing multi-granularity tagging is proposed to enhance protection against spatial and temporal memory violations without increasing memory overhead or system complexity [26]. ZOMETAG is a deterministic spatial safety solution, using MTE that partitions data memory into zones with unique tags, combining dual-layer isolation for efficient spatial safety protection with low overhead [19]. Additionally, Sfitag is proposed to optimize software fault isolation (SFI) in ARM kernel extensions with MTE, achieving efficient isolation by assigning different tag values to untrusted extensions and the core kernel [18]. A novel approach, called HeMate, is proposed to enhance heap security by isolating primitive data types using MTE, providing non-probabilistic protection[4].

These studies leverage hardware-level protection mechanisms to significantly enhance memory safety. Although these technologies have shown excellent performance in enhancing the security of C and C++ programs, their application in protecting Java heap memory remains in the exploratory stage and has yet to be widely implemented. Our work leverages the MTE hardware features to enhance the security of the Java language and has demonstrated excellent performance.

6.2 Native Language Based Approaches

Android 7 integrated AddressSanitizer (ASan) [20], which is a tool to effectively detect memory errors. To reduce the performance impact of ASan while maintaining a certain level of protection, GWP-ASan (Guarded Write-Protected AddressSanitizer) is proposed based on ASan [21]. GWP-ASan is a sampling-based tool for detecting memory safety errors in production environments , integrated into AOSP in Android 11. After ARM introduced the Memory Tagging Extension (MTE) feature, researchers further developed HWAddressSanitizer (HWASan) based on AddressSanitizer. HWASan

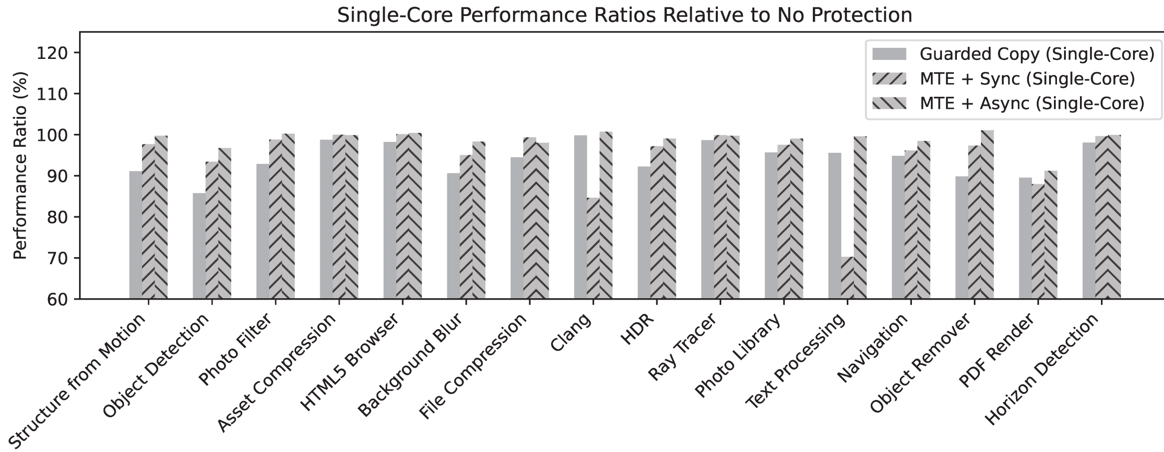

Figure 7. Relative Single-Core Performance of Sub-Items

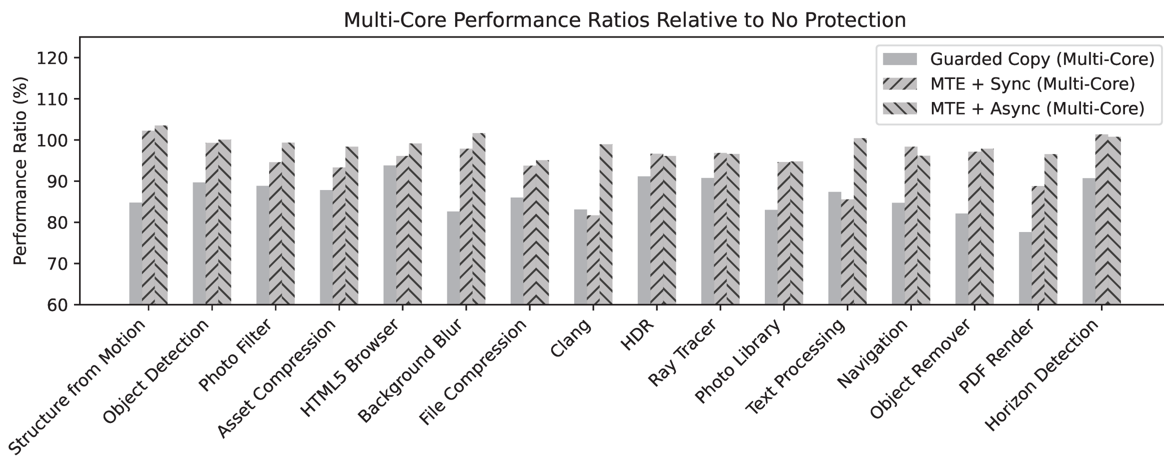

Figure 8. Relative Multi-Core Performance of Sub-Items

supports memory tagging technology at the hardware level for more efficient and low-overhead memory error detection.

Android also has protection mechanisms for kernel heap memory allocations. Android 12 introduced KFENCE (Kernel Electric Fence), a lightweight tool designed to detect kernel memory safety errors. It captures common memory safety issues such as out-of-bounds access and use-after-free by setting guard areas around allocated kernel heap objects [9].

These studies provide effective memory safety protection for native programming languages. However, they still have limitations in protecting Java heap memory. These tools are primarily targeted at memory errors in native code and offer limited protection for the Java environment.

6.3 JVM Runtime Based Approaches

At the runtime level, researchers have undertaken various efforts to mitigate the unsafe factors introduced by JNI. Safe-JNI is a framework ensuring type safety in heterogeneous programs involving Java and C components by addressing JNI vulnerabilities and adding static and dynamic checks [24]. A software-based fault isolation framework is proposed to place native code in a sandbox, allowing interaction with Java only through a controlled path [23]. This framework has two implementations: one integrated into the JVM and another external to it, trading some performance for JVM portability. JNICodejail is proposed to isolate native code to protect the JVM and Java memory from unauthorized modifications [7]. The Quarantine framework can mitigate memory errors in JNI applications by isolating unsafe objects in a special "JNI Space" and employing a read barrier and garbage collection mechanism [10].

These research efforts primarily focus on developing various frameworks and mechanisms to mitigate the risks posed by JNI and enhance the security and stability of applications. However, they have several limitations. First, most of these solutions rely heavily on runtime safety checks and isolation mechanisms, such as sandboxes and memory separation, which can effectively prevent unsafe access but do not help developers detect JNI-related vulnerabilities during the development phase. Second, many of these frameworks, like SafeJNI and Quarantine, introduce performance overhead due to the additional checks required to ensure safety. Third, some solutions, such as JNICodejail and SafeJNI, require modifications to the JVM or tight integration with it, adding complexity to the system. This increases the difficulty of maintaining and updating these frameworks, particularly when balancing performance, compatibility, and security.

In the current Android runtime, ART implements CheckJNI to detect and report JNI call errors, helping developers improve native code quality and stability [6]. CheckJNI identifies common errors such as negative array sizes, incorrect pointers and class names, improper JNI calls, and type safety issues. Guarded copy is a feature of CheckJNI, aimed at detecting out-of-bounds access in native code. However, as previously mentioned, it can only detect out-of-bounds write accesses, not out-of-bounds reads. And if the out-of-bounds access surpasses and skips the red zones, the error cannot be detected. Besides, this method has a significant impact on performance due to the introduced memory copying and synchronization. What's more, it can only detect whether an error has occurred, without providing detail information of the faulting location.

This paper, by leveraging hardware features, not only performs better in terms of efficiency but also has a stronger error detection capability.

7 Conclusion

We developed a novel memory safety protection method, MTE4JNI, aimed at safeguarding Java heap memory from illicit native code access. Leveraging ARM's Memory Tagging Extension (MTE), we integrated this approach into the Android runtime. Experimental results on real Android devices demonstrated that, compared to the currently employed guarded copy method, the proposed MTE4JNI method provided superior memory safety protection, while significantly reducing the runtime overhead on average by 11x and 27x for single-threaded and multi-threaded environments, respectively.

Acknowledgments

We thank all the reviewers for their insightful comments. This work was supported by the National Key Research and Development Program of China (No. 2022YFB3104502), National Natural Science Foundation of China (No. 62472330, No. 62272348), the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCH A202112 and OPPO Research Fund.

References

[1] 2020. *Memory Tagging Extension (MTE) in AArch64 Linux.* https://docs.kernel.org/arch/arm64/memory-tagging-extension.html Accessed: 2024-03-31.

[2] Android Open Source Project. 2023. Garbage Collection Debugging in ART and Dalvik. https://source.android.com/docs/core/runtime/gc-debug. Accessed: 2023-07-31.

[3] ARM. 2021. *Armv8.5-A Memory Tagging Extension.* White Paper. ARM Holdings. https://documentation-service.arm.com/static/624ea580caabfd7b3c13e23f Accessed: 2023-03-29.

[4] Yu-Chang Chen and Shih-Wei Li. 2024. HeMate: Enhancing Heap Security through Isolating Primitive Types with Arm Memory Tagging Extension. In *Proceedings of the 19th International Conference on Availability, Reliability and Security* (Vienna, Austria) *(ARES '24)*. Association for Computing Machinery, New York, NY, USA, Article 30, 11 pages. https://doi.org/10.1145/3664476.3664492

[5] Geekbench. 2023. *Geekbench 6 CPU Workloads.* https://www.geekbench.com/doc/geekbench6-cpu-workloads.pdf Accessed: 2023-03-29.

[6] Google Developers. 2011. *Debugging Android JNI with CheckJNI.* https://android-developers.googleblog.com/2011/07/debugging-android-jni-with-checkjni.html Accessed: 2024-03-31.

[7] Behnaz Hassanshahi and Roland H. C. Yap. 2013. JNICodejail: native code isolation for Java programs. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Stuttgart, Germany) *(PPPJ '13)*. Association for Computing Machinery, New York, NY, USA, 173–176. https://doi.org/10.1145/2500828.2500848

[8] Jiacheng Huang, Yunmo Zhang, Junqiao Qiu, Yu Liang, Rachata Ausavarungnirun, Qingan Li, and Chun Jason Xue. 2024. More Apps, Faster Hot-Launch on Mobile Devices via Fore/Background-aware GC-Swap Co-design. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 654–670. https://doi.org/10.1145/3620666.3651377

[9] Nikolai Joukov, Aditya Kashyap, Gopalan Sivathanu, and Erez Zadok. 2005. An electric fence for kernel buffers. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability* (Fairfax, VA, USA) *(StorageSS '05)*. Association for Computing Machinery, New York, NY, USA, 37–43. https://doi.org/10.1145/1103780.1103786

[10] Du Li and Witawas Srisa-an. 2011. Quarantine: a framework to mitigate memory errors in JNI applications. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. 1–10.

[11] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. 2022. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1901–1915. https://doi.org/10.1145/3548606.3560598

[12] Sheng Liang. 1999. *Java Native Interface: Programmer's Guide and Reference. 1st.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. isbn.

[13] Hans Liljestrand, Carlos Chinea, Rémi Denis-Courmont, Jan-Erik Ekberg, and N Asokan. 2022. Color My World: Deterministic Tagging for Memory Safety. *arXiv preprint arXiv:2204.03781* (2022).

[14] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. 2019. {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*. 177–194.

[15] Pascal Nasahl, Robert Schilling, and Stefan Mangard. 2021. Protecting Indirect Branches Against Fault Attacks Using ARM Pointer Authentication. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 68–79. https://doi.org/10.1109/HOST49136.2021.9702268

[16] Oracle. 2014. *Java Native Interface Specification*. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html Accessed: 2024-03-31.

[17] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: attacking ARM pointer authentication with speculative execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 685–698. https://doi.org/10.1145/3470496.3527429

[18] Jiwon Seo, Junseung You, Yungi Cho, Yeongpil Cho, Donghyun Kwon, and Yunheung Paek. 2023. Sfitag: Efficient Software Fault Isolation with Memory Tagging for ARM Kernel Extensions. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security* (Melbourne, VIC, Australia) *(ASIA CCS '23)*. Association for Computing Machinery, New York, NY, USA, 469–480. https://doi.org/10.1145/3579856.3590341

[19] Jiwon Seo, Junseung You, Donghyun Kwon, Yeongpil Cho, and Yunheung Paek. 2023. ZOMETAG: Zone-Based Memory Tagging for Fast, Deterministic Detection of Spatial Memory Violations on ARM. *IEEE Transactions on Information Forensics and Security* 18 (2023), 4915–4928. https://doi.org/10.1109/TIFS.2023.3299454

[20] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318.

[21] Kostya Serebryany, Chris Kennelly, Mitch Phillips, Matt Denton, Marco Elver, Alexander Potapenko, Matt Morehouse, Vlad Tsyrklevich, Christian Holler, Julian Lettner, David Kilzer, and Lander Brandt. 2024. GWP-ASan: Sampling-Based Detection of Memory-Safety Bugs in Production. (2024). arXiv:2311.09394 [cs.SE]

[22] Jeff Vander Stoep and Chong Zhang. 2019. *Queue the Hardening Enhancements*. https://security.googleblog.com/2019/05/queue-hardening-enhancements.html Accessed: 2024-03-31.

[23] Mengtao Sun, Gang Tan, Joseph Siefers, Bin Zeng, and Greg Morrisett. 2013. Bringing java's wild native world under control. *ACM Trans. Inf. Syst. Secur.* 16, 3, Article 9 (dec 2013), 28 pages. https://doi.org/10.1145/2535505

[24] Gang Tan, Andrew W Appel, Srimat Chakradhar, Anand Raghunathan, Srivaths Ravi, and Daniel Wang. 2006. Safe Java native interface. In *Proceedings of IEEE International Symposium on Secure Software Engineering*, Vol. 97. Citeseer, 106.

[25] Gang Tan and Jason Croft. 2008. An Empirical Security Study of the Native Code in the JDK.. In *Usenix Security Symposium*. 365–378.

[26] Martin Unterguggenberger, David Schrammel, Pascal Nasahl, Robert Schilling, Lukas Lamster, and Stefan Mangard. 2023. Multi-tag: A hardware-software co-design for memory safety based on multi-granular memory tagging. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*. 177–189.

[27] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. 2022. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 89–106. https://www.usenix.org/conference/usenixsecurity22/presentation/yoo

Received 2024-09-12; accepted 2024-11-04