AnBridge: Protecting On-Device AI with Android Virtualization Framework

Giorgio Farina*†, Raffaele Della Corte*, Aravind Machiry‡, Marcello Cinque*, Saurabh Bagchi‡
*Università degli Studi di Napoli Federico II, Naples, Italy
†Télécom Paris, Institut Polytechnique de Paris, Palaiseau, France
‡Purdue University, West Lafayette, IN, USA
giorgio.farina@telecom-paris.fr, {amachiry, sbagchi}@purdue.edu, {raffaele.dellacorte2, macinque}@unina.it,

***Abstract*—When deployed on-device, AI gets exposed to device-based attacks. To mitigate this threat, Android recently introduced the Android Virtualization Framework (AVF), enabling the creation of protected virtual machines (pVMs).**

However, the code running in the pVM is subject to a new programming model, making the porting of existing AI code in the pVM hard without a significant rearchitecting.

In this paper, we propose a new pVM programming model, and a related communication middleware, to simplify the migration of existing Android on-device AI code into pVMs. We evaluate the performance of our solution in terms of both latency and throughput, including workloads that access peripheral devices. Our results show modest overheads (2-14%) when pVM-resident native code invokes high-level Java functions running on the host. Finally, we discuss the practical deployment of our solution by applying it to two AI security-sensitive Android applications.

***Index Terms*—android, on-device-ai, security, privacy, porting, Android Virtualization Framework**

I. Introduction

Mobile apps are expected to generate over $935 billion in revenue in 2025, and Android dominates the smartphone market with over 72% share [10]. On-device data computation, as opposed to cloud computing, is becoming a popular solution for Android use-cases where low latency, low cost, and privacy safeguards are the primary concerns of a developer [30], [47]. On-device AI [42] is one of the main examples of this paradigm, as it usually processes sensitive user data for training or prediction.

However, local processing raises the challenge of protecting sensitive operations from device-based attacks [26], [42]. In this respect, a strong software adversary (e.g., rootkits) can compromise all software components, including the host OS, and can steal sensitive data, including AI models trained on it. This threat is exacerbated by the Android kernel that contains over 20 million lines of code and undergoes an astonishing rate of changes and rewrites.

As an example, let us consider the adoption of transfer learning to customize generic AI models to individual user needs. An attacker who gains access to the local model gains access to potentially sensitive content of the user ("*sensitive to the user*"), e.g., due to membership inference attacks [16], [17], [21]. At the same time, an increasing number of applications use proprietary AI-based models and databases. Hence, these companies desire to protect on-device computation from intellectual property theft to protect their market position ("*sensitive to the company*").

To protect sensitive workloads from device-based attacks, developers can consider to implement the sensitive logic as a Trusted Application (TA) [6], [24], [41] in a Trusted Execution Environment (TEE) on TrustZone [9]. However, mobile device vendors tend to be reluctant to make the secure world freely accessible to third-party developers[1], as the code running in TrustZone is assumed "trusted", and its vulnerabilities can jeopardize the execution of the other applications running outside the TEE [14], [23]. Recently, Android introduced[2] the Android Virtualization Framework (AVF) [8], which allows the developer to create a virtualization-based Trusted Execution Environment, which protects the enclosed execution from a compromised host (Figure 1). At the same time, the execution within the TEE is sandboxed and it cannot compromise the host. Essentially, Android TEEs are protected Virtual Machines (pVMs), running a limited-feature Linux OS (i.e., Microdroid), private to a specific application.

We argue that this emerging technology represents a valuable and concrete opportunity for researchers and practitioners to enhance the security and perceived trust of Android devices. We believe that the first step toward fostering this goal is to automate and simplify the porting of existing security-sensitive workloads to Android pVMs. This step raises several challenges, such as identifying security-sensitive functions [18], [19] and bridging heterogeneous programming interfaces and languages [5], [37], [43]. *In this paper, we address the problem of porting security-sensitive on-device AI workloads to pVMs to protect intellectual property and user confidentiality.* Across multiple abstraction layers, the enclosed native objects of AI libraries represent security-sensitive assets, as performance-critical workloads, i.e., training and inference, in Android apps are typically offloaded to native code for efficiency. Indeed, even when developers rely on high-level programming interfaces (e.g., Java/Kotlin), these interfaces ultimately invoke native implementations under the hood [45]. However, porting native code to Android pVMs is challenging without significant re-architecting, because this environment imposes

[1]The few applications accepted as TAs have to pass thorough vendor-specific testing, delaying the whole development process.

[2]Starting from Android 13, released in August 2022.

a fundamentally different programming model: unlike pVM native code, Android native code can interact with objects allocated and managed by the Java runtime.

In order to support the porting of Android app native code to the pVM, in this paper we propose a new *programming interface* between the Android native code running in the pVM and the Java code located on the host. The proposal is based on the popular Java Native Interface (JNI) [29], which plays a key role in the interaction between Java and native code within an Android app. Extending this interaction across the boundary between Android and the pVM, our solution allows developers to rely on a familiar and well-known interface, simplifying the porting of existing Android native code to the pVM.

To this aim, we extended AVF to support a new communication channel between Android and pVMs, which we call `AnBridge`, for **A**VF **n**ative **Bridge**. This channel allows the on-device native code running in the pVM to interact with Java code located on the host via JNI. By design, the new communication channel is faithful to two main principles. *(1) Mutually distrust model of AVF*: the security model of AVF is based on the principle of mutually distrust. In AVF, the code running in the pVM is not assumed "trusted", and the pVM security must not affect the security of Android or of another pVM. `AnBridge` cannot be used to over-privilege the host and other pVMs as its capabilities are still limited to those of the whole application. *(2) No shielding support needed:* when we add new logic inside a TEE (e.g., our communication channel), shielding support refers to the mechanisms required by the framework to sanitize interactions with the untrusted host. Contrary to other approaches that forward the system calls [12], [37], [43], our design implementation *does not require shielding support*, as it limits the visibility and capabilities of the host over the pVM address space by design. The interface never handles memory pointers belonging to the protected address space; therefore, no intermediate shielding layer is needed to validate or sanitize potentially malicious responses from the external OS.

We conducted an experimental analysis of `AnBridge`, evaluating its performance in different use cases (e.g., Java Custom Libraries, Java Third-Party Libraries and Java System Libraries), and comparing them against a VM-less implementation. Then, we apply our approach to a benchmark of two exemplary Android on-device AI apps to highlight the security implications and the practical usage of our solution. The applications emulate different scenarios where our solution is beneficial (i.e., intellectual property and user data protection), each one requiring specific security guarantees, i.e., data/code integrity and data confidentiality.

In summary, the paper provides the following contributions:

1) A novel programming interface for secure and confidential execution environment in Android. This is based on the standard JNI and makes use of the relatively recent AVF from Google, and avoids the limitations of using Trust-Zone, i.e., vendor-specific features and programming model, and static resource allocation.
2) The `AnBridge` to overcome the current limitations of AVF, i.e., limited non-standardized control and accessibility over the Java-side app, to make it as a programmer-friendly pVM, with no need for shielding support.
3) The evaluation of the performance, practical usage and security implications of the proposal on practical on-device AI apps. The `AnBridge` adds around 600 LOC to the Trusted Computing Base (TCB) of the pVM. The obtained results indicate that `AnBridge` performs better when the pVM runs code interacting with Java functions with high abstractions, e.g., Third-Party Libraries (TPL) or Java Custom Library, exhibiting 2-5% and 14% overhead in terms of network and image view latency, respectively, with respect to not using the pVM, and 3.5× disk throughput improvement with respect to *virtio*.

The rest of the paper is organized as follows. Section II presents the threat model considered in this study, including AVF, highlighting where our contributions take place. The design and architecture of our solution are presented in Section III and Section IV, respectively. Section V presents the obtained results in terms of proposal performance, while Section VI reports the practical security implications of our framework. Finally, Section VII discusses the related work, while Section X concludes the work.

II. Background

We discuss our threat model, how AVF can be adopted to get secuirty guarantees, and how our solution contributes to get protected on-device AI computation.

A. Threat Model

Our adversary model adheres to the one commonly assumed for TEE (e.g., TrustZone [12]) or CEE like SGX Enclaves [43]). The goal of the attacker is to leak or modify sensitive information from an Android application. In this perspective, an Android application is vulnerable to attacks on *code/data integrity*, and on *data confidentiality*, which may result in intellectual property and user data theft.

An application can be subject to both *in-app* and *OS attacks*, as depicted in Fig. 1. In-app attacks assume the possibility for an attacker to take control of the application by leveraging vulnerable code in the application or in a library used by it. OS attacks, also referred to as device-based attacks [42], instead assume that attackers can access the mobile device and gain access to its memory.

We target protection from OS attacks, which are stronger than in-app attacks, where a software-only adversary (e.g., rootkits) can compromise all software components, including the OS, except a small software/microcode TCB that configures the pVM, which is inherently trusted (i.e., pVM creation in Fig. 1).

Our threat model does *not* target code confidentiality at rest, denial-of-service or side-channel attacks, as well as faulty code running in the pVM. Precisely, malformed or adversarial

inputs from the host could exploit native vulnerabilities or enable model-extraction attacks; such input-driven threats are orthogonal to the migration problem addressed in this paper and fall outside our threat model. Moreover, we assume that the hypervisor layer (i.e., pKVM in Fig. 1, the code running at Exception Level 2) is trusted. Indeed, this layer is characterized by a small code base, and, more importantly, its size and code are not programmed to change. In addition, some existing studies are already working on hypervisor correctness and formal verification [22], [31], [35], [38], [48].

B. Android Virtualization Framework

In AVF, the TEE is represented by a protected Virtual Machine (pVM in Fig. 1) with the following security guarantees.

Run-time integrity/confidentiality: The contents of the memory owned by a pVM remain private unless the pVM explicitly shares it with another VM. Note that Android is confined within a VM (Host VM). VMs can not modify/influence each other's memory or CPU state without consent. The protection applies to both CPU and DMA accesses. Moreover, pVMs are provided with a stable sealing key, which is used for protecting persistent data.

Secure boot and Remote attestation: The integrity of the code booted into a pVM is verified. pVMs are provided with an attestation key, used for generating signatures that are verifiably produced by the pVM.

It is important to note that the availability of the pVMs is delegated to the host, instead, and thereby is not protected. Which means that attackers can easily start, shut-down, reboot, etc., pVMs in case of a compromised host. Instead, the code booted in the host and in the Exception Level 2, i.e., the hypervisor layer (pKVM in our study), is verified through the Android Verified Bootloader (AVB), which aims to assure the integrity of the software running on a device.

C. Protecting on-device AI computation

Protecting on-device AI computation from device-based attacks is paramount for the user and the developer. Our goal

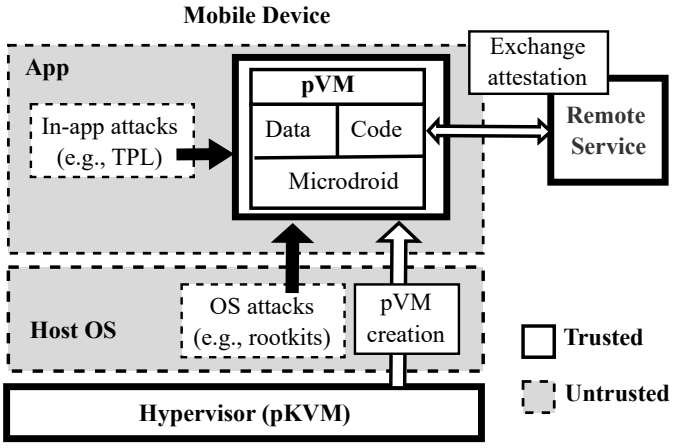

Fig. 1. Considered threat model under the Android Virtualization Framework (AVF). Our innovation allows sensitive parts of an app to be executed within the protected VM (pVM).

TABLE I
OPEN-SOURCE DNN LIBRARIES USED IN ANDROID APPS.

DNN Library	Java/Kotlin API	Core in Native code (JNI)	Native C/C++ API
TensorFlow	✓	✓	✓
TensorFlow Lite	✓	✓	✓
PyTorch	✓	✓	✓
ExecuTorch	✓	✓	✓
Paddle Lite	✓	✓	✓
OpenCV DNN	✓	✓	✓
DeepLearning4j	✓	✓	✗
ncnn	✗	✗	✓
Parrots (SenseTime)	✗	✗	✓

is to simplify the porting of existing AI security sensitive operations to a pVM in AVF. We analyzed the recent implementations of common open-source Deep Neural Network (DNN) libraries included in Android apps [26], [45] in Table I. From Table I, the developers have two possible programming models, which are basically the targeted programming models we deal within the paper: (1) The developer works with the DNN library from native code; (2) The developer calls the Java/Kotlin interface of the DNN library. However, in the last case, the Java/Kotlin layer is just a wrapper[3] that simplifies the interfacing with the native code. For instance, in TensorFlow Lite (TFLite) library, we have the interpreter and tensors allocated within native code, and the Java/Kotlin interface just simplifies the interaction with these native objects.

For instance, when we call *runSignature* (Listing 1) from Java to perform on-device AI training with TFLite, the Java implementation invokes several methods on Java wrapper objects, which just reflect the procedure call to the native objects. As an example, *getSignatureRunnerWrapper* calls a nested native function named *nativeGetSignatureRunner* (Listing 2), which returns an identifier of the native object.

Given that, we foresee a solution to automate the porting of selected Android native functions (e.g., *nativeGetSignature-Runner* in Listing 2) to an AVF-based CEE, preserving the integrity and confidentiality of code and data in execution. We do *not* target the automation of one-shot steps, including the step of remote attestation (Remote Service in Fig. 1), which assure the developer of the loaded code in the pVM (Exchange attestation in Fig. 1) and if the device is in potentially insecure state (i.e., unlocked devices).

III. PROPOSAL DESIGN

The cross-domain data access problem. Porting native implementations to a pVM is not straightforward, as the Android native code programming model within a pVM differs significantly from that of a regular Android application. A naive solution, i.e., simply compiling native code for a pVM and running it there, breaks down quickly for real-world workloads. The main reason is that Android native code often interacts with objects allocated and managed on the Java side. To assess this, we analyzed the frequency of cross-domain data access in the popular TensorFlow Lite library using the

[3]The only library completely in Java is CNNdroid [15], which has not been supported for nine years.

```
public void runSignature(
      Map<String, Object> inputs, Map<String,
      Object> outputs, String signatureKey) {
   ...
    NativeSignatureRunnerWrapper
         signatureRunnerWrapper =
    getSignatureRunnerWrapper(signatureKey);
    int subgraphIndex =
    signatureRunnerWrapper.getSubgraphIndex();
   ...
    signatureRunnerWrapper.invoke();
   ...
}
```

Listing 1. TFLite (Java code)

```
JNIEXPORT jlong JNICALL
nativeGetSignatureRunner(JNIEnv* env, jclass
    clazz, jlong handle, jstring signature_key) {
  ...
  if (interpreter == nullptr) return -1;
  const char* signature_key_ptr =
    env->GetStringUTFChars(signature_key, nullptr);
  SignatureRunner* runner =
  interpreter->GetSignatureRunner(signature_key_ptr);
  if (runner == nullptr) {
    // Release the memory before returning.
    env->ReleaseStringUTFChars(signature_key,
        signature_key_ptr);
    return -1;
  }
  // Release the memory before returning.
  env->ReleaseStringUTFChars(signature_key,
    signature_key_ptr);
  ...
}
```

Listing 2. TFLite (Native Code)

Tree-sitter parser [1]. Approximately 36% of native functions perform at least one cross-domain data access. In total, we observed 42 JNI method invocations, with an average of 2.21 JNI calls per interacting function. The code in Listing 2 shows one of the Android native functions in TFlite library, i.e., *nativeGetSignatureRunner*, accessing to the String Java object (jstring signature_key), by invoking *GetStringUTFChars* and releasing it by invoking *ReleaseStringUTFChars*.

To address cross-domain data access, one possible solution is copying the entire Java heap into the pVM for every native call, which is both *impractical and extremely inefficient*.

A more reasonable idea is to proactively determine which Java objects the native code might touch and copy only those into the pVM. Unfortunately, this requires accurate data-dependency analysis, which is notoriously difficult to perform in the presence of pointers [20]. Second, transferring Java objects through a communication channel like Binder (which is used in Android, as detailed later) requires the objects be parseable, which could require an additional manual effort for each specific workload and would be *application dependent*.

In addition, porting the execution of Java objects into the pVM is not a requirement in AI use cases. From a security perspective, our objective is to protect the AI processing pipeline (i.e., inference and training), which in Android ML stacks resides in native code for performance reasons. Java objects are mainly consumed, returned, or updated by AI processing and therefore do not require protection. The role of cross-domain access in AI use cases is thus functional rather than protective.

Our solution. Instead of copying everything up front, our key design choice is to leave Java objects on the host side, and facilitate transparent access from the pVM.

In Android apps, native code relies on the Java Native Interface to interact with Java objects, invoke Java methods, handle exceptions, and create new Java instances. By examining the *jni.h* library, it can be noted that all non-primitive types (such as jmethodID, jobject, jbyteArray, etc.) are opaque on the native side; native code can only interact with them through well-defined JNI APIs. In contrast, all primitive types have their counterparts in C++ and, once obtained with JNI (i.e., *GetStringUTFChars* in Listing 2), can be accessed directly by the native code without JNI.

Instead of copying everything up front, we leave the Java objects on the host side, relying on the fact that they cannot be accessed except through JNI calls. On the other hand, primitive types, being accessed directly, will be transferred from the host to the pVM and vice versa. In addition to tackling the cross-domain problem, our solution also has the following advantages:

- **Application independent.** Primitive objects, differently from arbitrary Java objects, are parcelable and can be easily transported, making the solution applicable to any Java code.
- **Low data transfer.** In most of the functions, only host Java references are transferred between the VM and the host, resulting in a reduced amount of data transfer. For example, in the code in Listing 2, the actual transfer occurs only when the *GetStringUTFChars* function is called.
- **No shielding support need.** The referenced Java objects of the JNI calls live host side in the address space of the app, so no shielding support is needed to limit the visibility of the host over the pVM address space. When we transfer primitive objects to the pVM, they are moved to the application in controlled buffers, and the native code receives a pointer to the buffer. Note that this solution does not ask to change the Android native code running in the pVM, as the native code reads objects already existing in the Java domain (e.g., the *signature_key_ptr* in Listing 2).

IV. Architecture and Implementation

We implement our solution using indirection, where we hook the JNI layer to jump to our `AnBridge`, which is a userspace library that can be easily integrated in the current Android applications without requiring modifications of Android OS. The architecture of our proposal is shown in Fig. 2. Currently, Android supports communication between the pVM and the host through Binder, which in turn relies on a *vsock* for data transfer. Binder offers the possibility to declare a set

of Remote Procedure Calls (RPC) that represent the protected payload exposed by the pVM (i.e., Protected Payload in Fig. 2) that can be invoked by the Java application running on the host. At the same time, the Java application can define and expose another set of functions that can be invoked (Binder callback) by the pVM. To support JNI calls within the pVM, we have two new entities, the `JNI Service`, instantiated in the Android application, that exposes a set of functions as binder callbacks and the `jni_stub.h` library, included by the protected payload in the pVM. The alternative to using Binder callbacks would have been to implement a custom shared memory between the host and the VM. However, setting up such shared memory would require modifications to Microdroid (i.e., the limited-feature Linux OS[4] running within the pVM), making the solution unsuitable for production use. Additionally, Binder is already based on *vsocks*, which leverage shared memory (via *virtio*) to optimize communication performance. The entire solution has been implemented in the Android Open Source Project (AOSP), and tested on a Google Pixel 7 Pro running Android 14. In the following, we describe the main components of our solution.

AIDL interface. We used the Android Interface Definition Language (AIDL) to define the interface of the `JNI Service`. AIDL allows defining the interface through which Inter-Process Communication (IPC) is performed between a client and a service. Given the AIDL interface definition, the Android Software Development Kit (SDK) generates a stub class in Java, which extends the Binder class, i.e., the core part of RPC support in Android. We extended the obtained stub class to provide the actual implementation of interface methods, which are exposed to clients for IPC.

JNI stub library. To enable the forwarding of JNI calls from the pVM to the host side, we create a stub of the *jni.h* library named `jni_stub.h`. The *jni.h* library defines a *JNIEnv* structure, i.e., a list of functions. Our stub library leaves the JNI interface unchanged, but replaces the available functions with stub functions. These functions transparently forward every JNI call to the host side, by using a reference to an instance of a `JNI Service` object running in the Android application. The `jni_stub.h` library should be used instead of the *jni.h* in the JNI code running in the pVM.

[4]Microdroid footprint is around 15 MB.

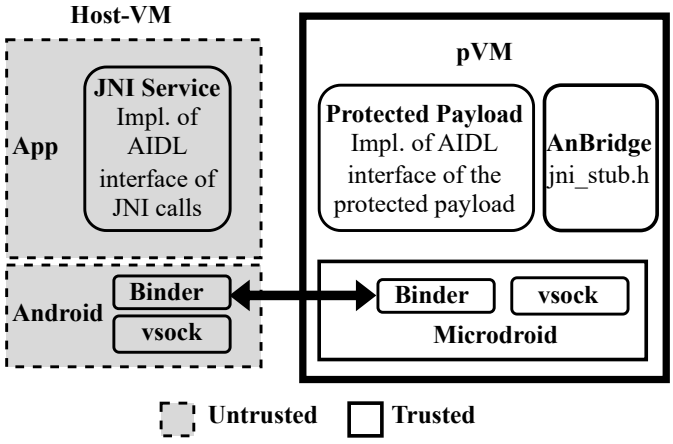

Fig. 2. Architecture of the proposed AVF-based solution.

JNI Service. We develop a `JNI Service` that can be executed in the context of any Android applications. The `JNI Service` receives the call from the `jni_stub.h` library and executes the real JNI call in the context of the Android application. The `JNI Service` AIDL interface was declared using the following criteria: each method in the interface has the same name of the related JNI function. The signature of each method has been modified as follows, instead: if the input or output parameters are primitive types or primitive type pointers, the corresponding primitive type in the interface was used (e.g., *jbyte** returned by *GetByteArrayRegion* becomes *byte[]*), while non-primitive types are passed by reference, represented as a 64-bit value (e.g., *jbyteArray* becomes *long* in the AIDL).

New transport channel. In Binder at most 40 KB can be transferred, which is prohibitive for real use cases.

In JNI, the only functions that can transfer large amount of data are the Get and Set functions of an array of primitive types (e.g., GetByteArrayRegion) and thereby only for these specific functions we supported a new transparent communication channel, i.e., `AnBridge`, which hides the data transfer Binder limitations. We open two new *vsock* (one for download and one for upload) in the VM at startup time. During the Get JNI functions, we have the following sequence of operations: (i) The `JNI Service` receives the RPC request from the function of the `jni_stub.h`; (ii) the `JNI Service` runs and detaches a new thread, which waits on the download socket; (iii) the control returns to the the `jni_stub.h`, which reads the data from the download socket and returns the data to the client of the `jni_stub.h` lib. Instead, during the Set JNI functions, the workflow is the following: (i) the function of the `jni_stub.h` starts and detaches a thread which waits to read from the socket and (ii) then calls the `JNI Service`; (iii) the `JNI Service` receives the RPC request from the pVM, reads the data from the download socket, and responds to the RPC. Noteworthy, when the data size does not exceed 40 KB, the channel uses only Binder for communication; otherwise, the *vsock*-based transfer is used.

V. Performance Evaluation

The evaluation aims to understand the performance of our solution, considering the following research questions:

- **RQ1. Channel performance of AnBridge:** How efficient is `AnBridge` at accessing/modifying real Java objects of different sizes located in the Android application?
- **RQ2. Performance of `AnBridge` in accessing peripherals:** What is the overhead of `AnBridge` when supporting workloads that access device peripherals?

The measurements have been performed on a Google Pixel 7 Pro, equipped with 11 GB RAM and 110 GB storage, and running Android 14 (Kernel 5.10 and December 2023

security patches) with the Virtualization package[5] enabled. We allocated a memory size of 250 MB for the pVM and that accommodated all the applications in our evaluation (one application is loaded at a time).

A. Evaluation Methodology

Get/Set Workloads. As a reminder, when used for JNI functions allowing to set and get data, such as *SetByteArrayRegion* and *GetByteArrayRegion* (*SetArray* and *GetArray* hereinafter), the proposed channel hides the data transfer cap of Binder (i.e., 40 KB) by transparently transferring the data on a raw *vsock*, when the data size is higher than 40 KB. Therefore, we use these functions to evaluate the performance of the channel. The *get-workload* and *set-workload* run within a pVM and use the *GetArray* JNI and *SetArray* JNI functions to access and modify a byte array field of a Java object located in an Android application, respectively.

Peripheral workloads. For the performance evaluation of our `AnBridge` solution in case of workloads accessing device peripherals, we prepared three different Android applications, each one accessing either network, disk or display. In the *network workload*, the Java code accesses the network to download digital content, such as an image. The *display workload* shows an image on the screen, while the *disk workload* accesses the Android application storage to obtain some contents (e.g., an encrypted AI model). The workloads have been developed considering three different programming styles: (i) *TPL*, where the developer makes use of a Third-Party Library to perform a specific operation, e.g., to make an HTTPS request or access to the disk; (ii) *CustomFunc*, where the operation is performed using a custom Java code defined in the application, e.g., a method of an existing Java user class that allows accessing the disk; (iii) *Java lib*, where we assume the developer accesses Java and Android libraries directly, e.g., accessing the disk by reading the Input Stream object or displaying an image with the ImageView object on the screen. It is important to note that to perform the overhead computation the portion of Java code to protect in the workloads has been translated to native *JNI-like* code. Therefore, we first run the workloads without using VMs as baseline, and then run them with our solution. Noteworthy, both workloads and baselines have the same code base, but the baselines run without any VM.

B. Channel Performance of `AnBridge`

We analyze the extent of the different components that add to the latency of a JNI function call using our channel, as well as the throughput of the channel. Fig. 3 reports the average latency of the *get-workload* (evaluated after 500 repetitions) when increasing the size of the array, and the main components affecting that latency. Instead, Fig. 4 shows the average throughput obtained in the same condition by the *get-workload* and the *set-workload*. It is important to note that when the *vsock* is used, the workflow described in Section IV takes place, i.e., the `JNI Service` first makes use of JNI to access the data (*JNI access data*), then setups the *vsock* on a new thread (*vsock startup*), and, finally, transfers the data via *vsock* (*vsock transfer*). Therefore, Fig. 3 also reports these contributions in addition to the total latency. In the following, we report the main findings of our analysis.

[5]Our setup uses the Virtualization module issued on Jan 31 2024.

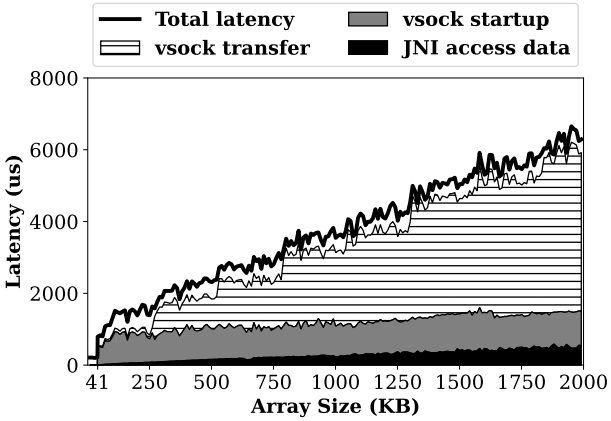

Fig. 3. GetArray latency breakdown with our channel.

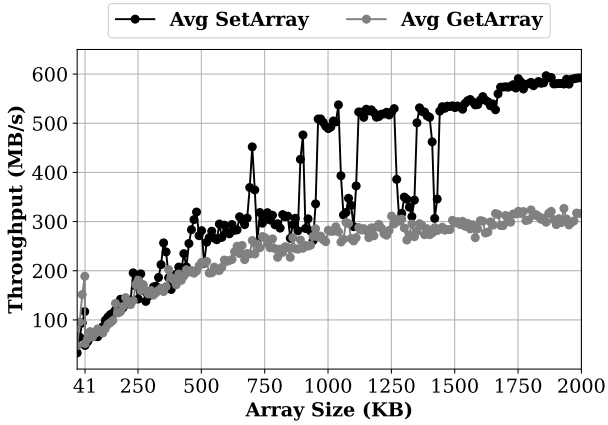

Fig. 4. Throughput of the proposed channel.

When the array size is smaller than 40 KB, the primary factor contributing to the latency is the Binder transfer/startup.
In this scenario, the `JNI Service` only makes use of JNI to access and return the data. We have four latency samples, i.e., 10 KB, 20 KB, 30 KB, 40 KB (that we interpolate in Fig. 3 and Fig. 4). The average latency (and throughput) is 210 μs (48 MB/s), 209 μs (98 MB/s), 197 μs (156 MB/s), 211 μs (194 MB/s), respectively; however, the `JNI Service` spends only 4 μs, 7 μs, 8 μs and 10 μs for each sample, respectively. Therefore, most of the time is spent in the Binder setup and data transfer.

When the array size is greater than 40 KB and lower than 350 KB, the *vsock* startup mainly affects the latency. Starting from 41 KB, i.e., when the *vsock* is used to transfer the data, the average latency start increasing. Indeed, the average latency increases to 792 μs (with 53 MB/s throughput),

reaching the 1525 μs (100 MB/s) and 2226 μs (200 MB/s) only when the payload size is higher than 150 KB and 350 KB, respectively. Fig. 3 highlights that this overhead is caused by the *vsock* startup, which introduces a constant penalty (i.e., around 800 μs) that mainly contributes to the latency when the data size is lower than 350 KB. It is important to note that this finding highlights the benefit of our design choice about using only Binder when data size does not exceed 40 KB.

When the array size is higher than 350 KB, the channel performance is mainly affected by the *vsock* transfer.

Fig. 3 highlights that starting from a data size of 350 KB, the main latency contribution is the *vsock* transfer time. Indeed, as discussed in the previous finding, the *vsock* startup is almost constant, irrespective of the data size. Therefore, its constant overhead does not increase when the array size increases. This benefits the throughput of our channel, which improves when increasing the array size as highlighted by Fig. 4. The obtained throughput results in Fig. 4 also show that there are different performance for the *SetArray* and *GetArray* functions. When the data size is lower than 40 KB, the *SetArray* exhibits a worse throughput, reaching at most 117 MB/s against the 200 MB/s of the *GetArray*. This overhead is attributed to Binder, which performs better when the array is in output rather than in input. Instead, the *SetArray* reaches higher throughput when the data size is higher than 40 KB (between 500 MB/s and 600 MB/s), rather than the *GetArray* (between 300 MB/s and 350 MB/s). To understand which is the factor to cause such a difference, we evaluated the speedup of the *vsock* during the different runs, and we discovered that the *vsock* provides a better average throughput (2$\times$ improvement) when data is transferred from the host to the pVM, over the pVM to host transfer.

RQ1. Channel performance of AnBridge

Finding: The performance of the AnBridge channel is affected by the size of the payload transferred between the pVM and the host. When the array size is smaller than 40 KB, the primary factor contributing to the latency is the Binder startup. When the array size is greater than 40 KB and lower than 350 KB, the vsock startup mainly affects the latency. When the array size if higher than 350 KB, the channel performance is mainly affected by the vsock transfer rate.

C. Performance of `AnBridge` in accessing peripherals

Fig. 5, Fig. 6 and Fig. 7, show the results obtained by using our solution to run the peripheral workloads and the baselines. Below, we present the results for each workload.

1) Network Access: In case of *Java lib*, the workload uses the Java library to download the data, i.e., reading in loop on an Input Stream[6] object, while for *TPL* it uses the OkHttp3 [40] library to make a HTTPS request to download the data. Instead, the *CustomFunc* assumes that the code to download the data is enclosed in method of a user Java object; therefore, the *JNI-like* code invokes that method. Fig. 5 shows the average latency to use the different solutions with and without our solution — *Baseline* is without our solution. We run the experiments with an access point that reaches 90 Mbps, with 500 repetitions for each setup.

The channel overhead in case of CustomFunc is higher for small data size, while it decreases when the size increases. The overhead of *CustomFunc* with respect to its baseline ranges between 19% and 29% for lower package size (40 KB and 10 KB, respectively). However, for data size greater than 40 KB the response time is just 2-5% higher than its baseline (100 KB and 1 MB package size, respectively). This is an expected result since, as seen in Section V-B, for small data size the *vsock* and Binder startup strongly affect the JNI function call latency, but that diminishes when the package size increases.

The increment of the data size does not lower the overhead in the case of Java lib. In contrast to *CustomFunc*, the *Java lib* overhead is still high for data size greater than 40 KB. Indeed, *Java lib* reaches 31% overhead for 10 KB, and it still 24% for 1 MB. It is important to note that in this case the Java code reads 8 KB at a time from the Input Stream. Therefore, our channel does not use the *vsock*, since the data transferred at each round is lower than 40 KB. Moreover, given the limit on the Input Stream, multiple JNI invocations are needed to transfer the total data size. Thus, the higher the total data size, the higher the number of JNI calls made by the pVM code, which leads to an increased overhead compared to *CustomFunc*.

When using a TPL, our solution introduces a negligible overhead irrespective of the data size. The response times of *TPL* are 2-3% higher than the baseline. As in the case of *CustomFunc*, the number of JNI function calls does not depend on the size of the data, since the pVM code only calls a single *TPL* function to transfer the entire data volume. Therefore, the data size does not affect the obtained overhead.

2) Disk Access: For this workload, the Java code makes use of the *common-utils* library [3] for the *TPL* case, while it leverages the Java library primitives for file reading in the *Java lib* case. The considered scenario consists in reading a 50 MB[7] file located in the application storage. In addition to *Java lib* and *TPL*, we also consider *virtio* as a baseline. *virtio* represents a solution to provide storage to a virtual machine, i.e., access to an external file located in the host, and thereby it represents a good metric of comparison when measuring the access from the pVM to the storage of the application. Differently from the network workload, the size of the disk IO buffer can be higher than 8 KB. Therefore, Fig. 6 shows how the throughput (MB/s) of the *Java lib* at varying buffer sizes. On the other hand, the average performance of the baselines, *TPL* and *virt-io pVM* are shown as horizontal lines in Fig. 6, as we observed the same performance for the *Java lib* baseline irrespective of the data size, while for the others it is not possible to vary the buffer size. The file is read sequentially; we repeat each experiment

[6]Note that at most 8,192 bytes can be read from a Java Input Stream; therefore, we used this value as buffer size.

[7]The size has been selected considering the one used by the Microdroid-BenchmarkApp of AVF to estimate the throughput of *virtio*.

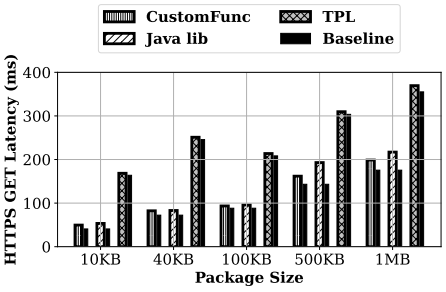

Fig. 5. Application accessing network: Data request latency.

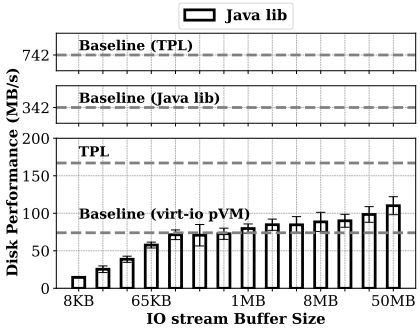

Fig. 6. Application accessing disk: File reading performance.

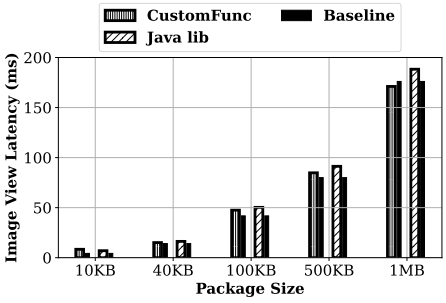

Fig. 7. Application accessing display: Image view latency.

20 times, flushing the memory at each run to ensure that no data is cached.

Understandably, the throughput of our solution is significantly below those of the Java side baselines (no traversal of trusted-untrusted boundary). However, our solution outperforms *virtio* when used to access a file available in the Android application storage. As expected, the direct access to disk from the vanilla Android app reaches high IO throughput when compared to accessing the disk from a pVM. The baselines, reach a throughput of 742 MB/s and 342 MB/s, respectively for TPL and Java lib, which are around 10× and 5× faster than *virt-io pVM* (74 MB/s), i.e., our pVM side baseline. Indeed, the *TPL* reaches a throughput of 167 MB/s, i.e., 3.5× faster than *virt-io pVM*. On the other hand, *Java lib* exhibits a throughput that ranges between 20 MB/s (8 KB buffer size) and 110 MB/s (50 MB buffer size), outperforming *virt-io pVM* when the buffer size is greater than 130 KB. It is important to note that the different performance exhibited by the *Java lib* at varying the buffer size is mainly due to the number of JNI calls required to perform the whole file read operation. Indeed, higher the buffer size, lower the amount of required JNI function calls. For example, a buffer size of 8 KB requires around 6,400 JNI calls against the 400 requested with a buffer size of 130 KB.

3) Display Access: In this workload, we consider that the update of the ImageView is triggered. Note that this operation can be executed only by the main thread of the Android app. Therefore, we consider two scenarios. In the first scenario (*Java lib*), the code that we want to port to the pVM is located in the main thread of the app. In this case, we assume the code makes use of the Android library, which consists in defining a Bitmap object and setting the ImageView. Instead, in the second scenario, *CustomFunc*, we assume that ImageView update is invoked by a secondary thread through a function available on the main thread. The Java code executed by the main thread (*Java lib*) is used as a baseline for both cases, as they perform similarly. We use the same images used for evaluating the network workload. We repeat each run 10 times and measure the average latency to show the image, which are shown in Fig. 7. Contrary to the network and disk evaluations, we do not take in consideration the usage of a *TPL* since Android (*Java lib*) itself provides a high-level abstraction of the ImageView.

The overhead introduced by our solution to show images on the screen is higher for small image size. *Java lib* and *CustomFunc* spent 7 ms and 8.4 ms, respectively, to display an image of 10 KB, while on the host side the same operation takes 4.3 ms on average. Instead, when the image size is higher than 40 KB, both *Java lib* and *CustomFunc* are around 1.1× slower than the baseline on average.

For example, *Java lib* and *CustomFunc* spent 50.3 ms and 47.5 ms, respectively, to show a 100 KB image, i.e., 1.2× and 1.1× slower than the baseline (41.8 ms), respectively.

RQ2. Performance of AnBridge in accessing peripherals

Finding: Our results indicate that `AnBridge` performs better when the pVM runs code interacting with Java functions with high abstractions, e.g., TPL, exhibiting 2-5% and 14% overhead in terms of network and image view latency, respectively, with respect to not using the pVM, and 3.5× disk throughput improvement with respect to accessing storage from the pVM with *virtio*.

D. Discussion

Our approach enables the protected payload to interoperate with the Java code running in the Android application in a flexible way. It enables the native code in the pVM to seamlessly transfer data between the host and the pVM, leveraging *vsock* when needed and to seamlessly communicate with the Java code through the standard *JNI-like* interface. According to the obtained results, our approach performs better when the code running within the pVM interacts with Java functions with sufficient abstractions, like in the case of *TPL* and custom function usage. Indeed, when *TPL* or custom functions are invoked via JNI, the proposal reaches good performance, i.e., 2-5% and 14% overhead in terms of network and image view latency[8], respectively, with respect to *Java lib*, and 3.5× disk throughput improvement with respect to *virt-io pVM*. It is important to note that the data transfer overhead cannot be eliminated since the security-critical portion of the application has to execute within the pVM. Moreover, as observed from the results in both Section V-B and Section V-C, setting the

[8]The overhead for custom functions refers to data size higher than 40 KB.

stream buffer size to small values, e.g., values equal or lower than 40 KB for network and screen access, and lower than 130 KB for disk access, can lead to an increase of the overhead. However, we argue that the usage of such data sizes is mainly required when the code running in the pVM needs fine-grained data access, e.g., to process small batches of data at time, which are outlier use cases. The obtained performance in that case depends on the overhead of each JNI call and the large number of JNI calls that must be made. This was seen in the case of the *Java lib* in the network workload, where the total data transfer requested multiple JNI invocations that led to a 20% overhead on average.

In conclusion, workloads that rely on frequent fine-grained Java – native code interactions incur higher overhead due to repeated cross-boundary transitions (JNI + transport), whereas coarse-grained or library-level calls amortize this cost. However, this reflects a design trade-off in AnBridge: we prioritize compatibility with existing Android AVF abstractions (i.e., Binder and vsock) to provide a production-ready solution. We agree that alternative implementations could further reduce overhead for fine-grained patterns, for example by introducing specialized cross-domain protocols based on shared memory between the pVM and the host runtime but they would need changes in AVF. Exploring such designs, relaxing compatibility with existing AVF abstractions, is an important direction for future work.

VI. Security Implications

We discuss the practical security implications of our proposal and evaluate its performance in the context of on-device AI applications, considering the following research questions:

- **RQ3. Securing applications**: How and in which terms AVF and `AnBridge` can be used to protect on-device AI apps?
- **RQ4. Performance penalty**: Which is the performance penalty of the proposal in the considered use cases?
- **RQ5. TCB size**: How does `AnBridge` enlarge the TCB?
- **RQ6. Shielding support**: How does `AnBridge` influence the TCB and the necessity of shielding mechanisms?

Note that we used the same device and setup indicated at the beginning of Section V.

A. Evaluation Methodology

We manually created two Android applications[9]. The applications aim to emulate two main scenarios where our solution is beneficial: protecting on-device AI Intellectual Property (IP) and protecting user data. Precisely, the evaluated applications were designed to capture distinct and realistic security-sensitive AI use cases in Android. `App#1` focuses on safeguarding training data and model confidentiality to protect user privacy and intellectual property, while `App#2` targets model integrity and execution correctness against tampering. Together, these scenarios reflect common threat categories in on-device AI deployments.

Their characteristics are shown in Table II. The last column represents the size of the APK and within parentheses, the cumulative size of the functions protected in `AnBridge`.

App#1 - Protecting IP/User Data of Local AI Models: When AI models are deployed on-device, the developers desire to protect their intellectual property and, thereby, the confidentiality of their AI models, as well as the confidentiality of the collected user data, used to fit user habits [42]. We built an application using an open-source AI Image classification model [13] (146 KB): given an image taken from the camera or the gallery, the application leverages the model to classify the image based on the kind of food it detects in the image. The kind of food detected from the image, associated with other info (e.g., the time), populates a dataset at run-time. A decision tree is trained on the dataset to predict user food habits. The application requires to guarantee that no unauthorized access can be made on the AI model, the collected dataset, and the trained model, i.e., data confidentiality, to protect the IP and user data they embed. The application is characterized by four main *security-sensitive* functions to be protected, i.e., `GetModel`, `ImageClassification`, `CollectFitUserData`, and `PredictUserHabits`, dedicated to retrieving the local model, performing the image classification, fitting user data, and predict user habits, respectively.

App#2 - Enforcing Local Policies: On-device AI is often used to enforce safety or security checks [46]. In this application we consider the social network scenario, where the application performs checks on the comments the user wants to publish [39]. The developer wants only content that complies with the policies to be published, while an attacker might attempt to bypass these checks. An attacker may choose to analyze the model (i.e., exploiting data confidentiality) to understand which words or combinations of words might not be detected, modify the model's configuration (i.e., exploiting data integrity) to allow disallowed posts. Therefore, the application requires protection of the on-device AI model (data integrity and confidentiality). In this perspective, the application has two main functions to protect, i.e., `GetModel` and `PublishComment`. The `GetModel` function is similar to the one used for `App#1`. The only difference is related to the model, which is more complex (26 MB) in this application and includes also a dictionary of words (126 KB). The `PublishComment` function leverages the model to verify if a comment is compliant with the application policy.

For each application, we created two versions: (i) a non-protected version, representing our baseline, and (ii) a protected version, encompassing the stated security-critical functions, protected with our approach. The two versions allow us to highlight the differences in terms of security and overhead.

Non-protected version: We assume that the non-protected version uses encryption to protect their data from static anal-

[9]Note that AVF is a recent technology and there are no use cases of applications available online. The official Android documentation supports only the isolated compilation use case [11], where AVF is used to protect the isolated compilation task, and not from an application developer perspective.

TABLE II
APPLICATIONS FOR SECURITY ANALYSIS. THIS ALSO SHOWS THEIR PERFORMANCE WITH ANBRIDGE PROTECTION AND WITHOUT PROTECTION.

App	Function	Latencies "median"		LOC		Apk
		Non Protected	Protected	Java	Native	(vm.so)
#1	GetModel	2 ms	6 ms	270	771	20 MB (2.7 MB)
	ImageClassification	2.5 ms	2.6 ms			
	CollectFitUserData	10 ms	9 ms			
	PredictUserHabits	305 us	817 us			
#2	GetModel	207 ms	765 ms	395	519	39 MB (2.7 MB)
	PublishComment	407 ms	410 ms			

ysis attacks [26], which means keeping the assets encrypted[10] when they are stored on the disk and be decrypted at runtime. To obtain the key to decrypt the model, the non-protected versions solely rely on HTTPS and do not add an additional encryption layer in the communication with the server, since we are assuming that the host can be compromised as per our attack model (i.e., *device-based* attacks); therefore, such a measure would be pointless.

Protected version: On the other side, the protected version of the applications assumes the attacker has full control of the app memory space, as in our threat model. The protected version makes use of the well-known technique of *remote attestation* [28] to enable the remote server of the developer to verify the integrity and authenticity of the code running in the pVM. After the remote attestation, the remote server and the pVM have a common shared secret.

B. Securing Applications

From a security standpoint, we highlight the main differences between a non-protected and protected version of the two target applications. As also mentioned in Section II, we do not target the automation of one-shot steps such the remote attestation. In the protected version of our apps, the step of remote attestation is performed by the `GetModel` function (Remote Service in Fig. 1), allowing the developer to certify the loaded code in the pVM (Exchange attestation in Fig. 1) and certify that the device is in a secure state (i.e., unlocked devices). In `GetModel`, the model to be protected is distributed as a TensorFlow Lite file embedded in the APK in encrypted form. The native method `GetModel` uses the JNI interface to load the model from the application storage and perform an HTTPS request to retrieve the decryption key from a remote server.

In the **non-protected** version of `App#1` and `App#2`, once the encrypted model is retrieved and decrypted, it is mapped into memory in the app's address space. This provides no runtime protection, as the model can be accessed by malicious code at application or OS level. Moreover, training and prediction run in the untrusted host environment, where collected data and model parameters remain exposed in memory.

Instead, in the **protected** version of the applications, the model is loaded into the pVM after remote attestation. Both the key and the decrypted model are then stored on the pVM's encrypted disk. The model remains mapped in protected memory (i.e., `memmap`) for the app's entire lifecycle, ensuring it is inaccessible from outside the pVM and securing it against runtime attacks. Moreover, data collection, training, and inference are executed entirely within the pVM, ensuring code/data integrity and data confidentiality, granting intellectual property and user data protection.

[10]We used AES-128 with a secret and an initial vector of 16 bytes.

RQ3. Securing applications

Finding: The proposal can be used to host and protect Android on-device AI use cases, including the protection of confidentiality and integrity of AI model computations from device-based attacks.

C. Performance Penalty

We evaluate the latency of using the functionalities of the applications that need protection in both protected and non-protected versions. In both versions, the security-sensitive code is included as native code (i.e., C++) that uses JNI to interoperate with the Java-side. The results are summarized in TABLE II.

The functions of `App#1` show low latencies ($< 10ms$), so we average over a large number of observations (i.e., 500). The results for `CollectFitUserData` considers the training on a user dataset of 500 samples, while for `ImageClassification` the latency of classification is identical across images. The functions of `App#2` show higher latencies, so we average over a smaller number of observations (100). `PublishComment` does not show significant differences when we change the size (until 2,000 chars) of the string to publish, so we setup the comment to a word lower than 100 chars. We calculated the overhead as $((\text{Lat}_{\text{Protected}} - \text{Lat}_{\text{NonProtected}}) \times 100)/\text{Lat}_{\text{Protected}}$.

The results indicate that in the considered scenarios the overhead is approximately 4%, 0.74% for the `ImageClassification` and `PublishComment`, functions, respectively. Instead, the overhead of `PredictUserHabits` is higher, i.e., $2.5\times$. This is caused by the extremely low-latency nature of the function itself; the `AnBridge` call brings a significant overhead. In contrast, in `CollectFitUserData` there is even a performance gain, i.e., 10%.

The code versions (non-protected and protected) of `CollectFitUserData` run the same code. This brings to conclude that the training function spends less time in the pVM rather than in the Android app address space, probably due to the low-featured kernel hosted in the pVM. The `GetModel` function exhibits a high overhead in both `App#1` and `App#2`, i.e., $3\times$ and $3.5\times$, respectively, since the data is copied to the encrypted disk during the loading. However, this latency occurs only the first time the app is opened, as the next times the content will be retrieved directly from the pVM encrypted disk and not from the application storage. Noteworthy, the latency values of the `GetModel` function vary across the two applications due to the higher complexity of the model used by `App#2`.

RQ4. Performance penalty

Finding: Across the evaluated scenarios, most functions incur only modest overheads (i.e., 4%, 0.74%) when executed in the pVM. Higher overheads (2.5x) are observed for low-latency tasks involving JNI transitions. Notably, in some cases, execution in the pVM yields a performance gain (i.e., 10%), as the lighter pVM kernel offsets the cost of isolation with improved efficiency.

D. TCB Size.

We analyze the impact of our implementation on the TCB. A smaller TCB is generally easier to review or formally verify, and it is assumed to contain fewer vulnerabilities. TABLE II also reports: (i) the lines of code (LOC) for both Java and native components, and (ii) the size of the Android Package Kit (APK), i.e., the installed application. Additionally, TABLE II includes in parentheses the size of the shared library (`vm.so`) that is installed at runtime inside a pVM running Microdroid, i.e., the protected payload. The protected payloads bundle utility libraries (e.g., TensorFlow, OpenSSL) and the `AnBridge` interface library `jni_stub.a`.

Our `jni_stub.a` adds at most 540 KB to the TCB. To better estimate the size contribution of our `AnBridge` interface, we built a stripped-down "implementation-less" version of the library, where all `AnBridge` functions have empty bodies (reducing the size of `jni_stub.a` from 540 KB to 60 KB). When linking this version into the protected payloads, the sizes of `vm.so` for `App#1`, and `App#2` decrease by 25.2 KB and 25.4 KB, respectively (i.e., around 2.67 MB, 2.67 MB, as final size, respectively). This corresponds to a relative TCB overhead of approximately 0.15% (25.2 KB/2.7 MB) for each of `App#1` and `App#2`.

RQ5. TCB size

Finding: Our AnBridge library contributes at most 540KB to the TCB. For the applications studied, after the linking stage, this corresponds to a relative TCB overhead of only 0.15% for `App#1` and `App#2`.

E. Shielding Support.

When a solution extends the boundary of a TEE, its implementation can weaken its security guarantees. We therefore evaluated whether shielding support was needed to protect the pVM from potential host threats introduced by our design. To clarify what we mean by shielding support, consider prior work that adopts a delegate-rather-than-emulate strategy [12], [37] to provide POSIX interfaces inside TEE environments (e.g., TrustZone or Intel SGX). Such systems require intermediate layers to constrain host visibility into the protected environment. For example, when a TEE delegates a system call to the host by passing a pointer to a buffer, a shielding layer must restrict the host's access to that buffer (e.g., via bounce buffers) to prevent Iago attacks [7]. Unlike these systems, AnBridge does not require additional shielding layers in the pVM. Java objects remain in the host environment, and only explicitly marshalled raw data are transferred to designated buffers. Therefore, once enabled, AnBridge does not require intermediate shielding layers to constrain host visibility.

RQ6. Shielding support

Finding: AnBridge does not need new shielding layers into the pVM to constrain host visibility across pVM memory.

VII. Related Work

Enabling apps to isolate parts of their execution from the untrusted host is a shared vision and issue in cloud computing [4], [37], [43], IoT [12] and mobile platforms [14], [33], [36]. The logic to protect depends on the context and the valuable assets. Several studies are proposed to identify the minimum set of functionalities to place in the sandbox [18], [34]. Since our solution centers on the design of a new interface, we discuss prior studies that provide a new interface in isolated execution environments. While other studies focus on TrustZone and Intel SGX, providing a system-level abstraction to enable the transparent porting, our work provides a new abstraction interface to address the limitations of the current programming model of AVF. In addition, to the best of our knowledge, this is the first work that address AVF programming limitations. Some of the existing AVF studies mainly focus on the hypervisor correctness and formal verification [22], [31], [35], [38], [48], secure access of a pVM to the peripherals [25], [27], or pVM optimizations to mitigate the overhead of *virtio* in I/O intensive workloads [44].

Below, we compare the properties of our abstraction interface with similar studies in the context of isolated execution environments. Several studies [4], [5], [12], [37], [43] provided a low-level interface so that a complete application can live inside an confidential execution environment. Library OSes provide an abstraction of a virtualized process namespace to application code. PANOPLY [37] and TrustShadow [12] expose the standard POSIX abstractions to application logic with a reduced TCB adopting a *delegate-rather than-emulate* strategy. The partitioned code does not get its own virtualized namespace but shares it with its host Linux process. Tarnhelm [33] adopts the same strategy but limits its scope to code confidentiality, leaving the data unprotected. The problem with the *delegate-rather-than-emulate* strategy of system services is that the sandbox cannot protect an application that requires functionality from the OS. Iago attacks [7] are semantic attacks from the untrusted OS against the application, where an unchecked system call return value or effect forces the protected code to disclose or modify sensitive sandbox data. Iago attacks can be subtle and hard to comprehensively detect, at least with the current POSIX or Linux system call table interfaces. Thus, any sandbox framework usually provides complex and error-prone intermediate layers, i.e., *shielding support*, to validate or reject inputs from the untrusted OS. A real limitation of designing shielding support is that more complex system calls, such as *ioctl* and *fcntl*, have different behaviors according to subcommands. A marshalling code for each request/command must be prepared separately according to the specifications of the subcommands, i.e., a *system-call/workload dependent* solution.

Moreover, Lind et al. [18] observed that bringing an entire app into a sandbox can be useless, as only specific parts of the code need protection. Further, it can also be risky, as it can bloat the sandbox with untrusted code (e.g., untrusted TPL).

Differently form these studies, our abstraction is workload independent and requires no shielding support.

VIII. Threats to Validity

Our work introduces a new communication channel aimed at supporting a developer-friendly programming model within the pVM. While the proposed architecture is effective in demonstrating feasibility, several assumptions and limitations could affect its broader applicability.

Access to peripherals: Our design assumes that the pVM has secure access to selected peripherals. While secure end-to-end communication (e.g., for networking) can be achieved by layering additional encryption protocols, enabling secure and efficient access to more complex peripherals, such as touch-screens or GPUs, poses greater challenges. In our prototype, we adopt a temporary peripheral donation model inspired by [25], where exclusive access is granted to the pVM for the duration of its usage. This design inherently limits concurrent access to the peripheral and may impact system-wide resource utilization and responsiveness. More advanced sharing models (e.g., mediated passthrough or paravirtualization) are out of scope but could be explored in future work.

TCB reduction: Our approach assumes the availability of mechanisms to reduce the TCB size to fit within the pVM. Specifically, we expect that users can selectively include only the security-critical components, and eliminate unused or vulnerable libraries. However, such a process may require significant manual effort or advanced tooling (e.g., static/dynamic analysis, library hardening, API boundary enforcement), and its complexity may hinder adoption in practice. Automating or integrating TCB minimization into the development toolchain remains an open challenge.

JNI threat: Native code in the pVM may include untrusted malicious libraries. Such untrusted code in the pVM can use JNI in a malicious way to access other app's capabilities, but it cannot attack directly the host operating system (e.g., in case of rootkits int he pVM) thanks to *the mutually distrust model* of AVF. New checks can be performed on transactions in the `JNI Service` by limiting the classes and methods that can be accessed [2], [32].

IX. Limitations and future directions

Automation of the approach: Our solution is only partially automated and is currently designed for developers, assuming the availability of source code. The required manual effort is as follows. First, the developer selects the files that contain native functions (these files are usually easy to identify, as Java and native code are typically organized in separate directories). In these files, instead of including `jni.h`, they must include our stub `jni_stub.h`. After that, the developer compiles the files for the pVM, selecting the functions that will be exposed and accessed from the host. Specifically, the developer selects the native functions exposed to Java code, which are easy to identify because they start with `JNIEXPORT`.

Input attack surface: Malformed or adversarial inputs originating from the host could exploit vulnerabilities in native code or enable model-extraction attacks. Such input-driven attacks are orthogonal to the isolation problem addressed by AnBridge and fall outside our threat model. Future research may focus on evaluating the security of assets in the pVM when exposed to malicious inputs.

Portability to other use cases: Our system enables the reuse of Android native code within the pVM with minimal modifications, which is effective for our target domain, i.e., on-device AI. While our design choices are tailored to the AI use case (for example, leaving Java objects accessed from native code outside the pVM) many other case studies remain to be explored. For instance, security-sensitive code paths in Android applications may involve higher-level Java or Kotlin libraries, which are not directly portable to the pVM environment.

X. Conclusion

We presented `AnBridge`, a new abstraction interface to aid the flexible development of protected on-device AI payloads in Android Virtualization Framework. The main idea behind our solution is to take inspiration from the in-app interaction between Java and native code, extending this interaction across the boundary between Android and the pVM. We provided a thorough experimental analysis by using workloads accessing peripherals and two full-fledged apps. `AnBridge` is a lightweight library that allows the protected payload to flexibly interoperate and reuse Java objects located in the application, without exposing the binder limitations to the developer. `AnBridge` performs better when the code running within the pVM interacts with Java functions with high-level abstractions, i.e., 2-5% and 14% overhead in terms of network and image view latency, and 3.5× disk throughput improvement with respect to *virtio*. It enlarges the TCB size by only 0.15%.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments, which greatly improved our paper. This research was partly supported by the National Science Foundation (NSF) under Grants CNS-2340548 and CNS-2333487 and the Army Research Lab (ARL) under Contract number W911NF-2020-221. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

This work was partially supported by the SERENA-IIoT project, which has been funded by EU - NGEU, Mission 4 Component 1, CUP J53D23007090006, under the PRIN 2022 (MUR -*Ministero dell'Università e della Ricerca*) program (project code 2022CN4EBH).

REFERENCES

[1] Github repository. tree-sitter, a parser generator tool and incremental-parsing library. https://github.com/tree-sitter/tree-sitter, 2025.

[2] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, Giovanni Vigna, et al. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *The Network and Distributed System Security Symposium 2016*, pages 1–15, 2016.

[3] Apache. Commons io library. https://github.com/apache/commons-io/tree/master, 2024.

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. {SCONE}: Secure linux containers with intel {SGX}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[5] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–26, 2015.

[6] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[7] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. *ACM SIGARCH Computer Architecture News*, 41(1):253–264, 2013.

[8] Android Documentation. Android virtualization framework). https://source.android.com/docs/core/virtualization, 2024.

[9] Arm Documentation. Arm TrustZone for Cortex-A. https://www.arm.com/technologies/trustzone-for-cortex-a, 2024.

[10] Statcounter GlobalStats. Mobile operating system market share worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide, 2025.

[11] Google. Android virtualization framework: Use cases. https://source.android.com/docs/core/virtualization/usecases, 21 June 2024.

[12] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 488–501, 2017.

[13] IJ-Apps. Image classification app with custom tensorflow model. https://github.com/IJ-Apps/Image-Classification-App-with-Custom-TensorFlow-Model, 2024.

[14] Jinsoo Jang, Changho Choi, Jaehyuk Lee, Nohyun Kwak, Seongman Lee, Yeseul Choi, and Brent Byunghoon Kang. Privatezone: Providing a private execution environment using arm trustzone. *IEEE Transactions on Dependable and Secure Computing*, 15(5):797–810, 2018.

[15] Seyyed Salar Latifi Oskouei, Hossein Golestani, Matin Hashemi, and Soheil Ghiasi. Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android. In *Proceedings of the 2016 ACM on Multimedia Conference*, MM '16, pages 1201–1205, 2016.

[16] Klas Leino and Matt Fredrikson. Stolen memories: Leveraging model memorization for calibrated {White-Box} membership inference. In *29th USENIX security symposium (USENIX Security 20)*, pages 1605–1622, 2020.

[17] Jiacheng Li, Ninghui Li, and Bruno Ribeiro. {MIST}: Defending against membership inference attacks through {Membership-Invariant} subspace training. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2387–2404, 2024.

[18] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for intel {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, 2017.

[19] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2359–2371, 2017.

[20] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2359–2371, New York, NY, USA, 2017. Association for Computing Machinery.

[21] Yugeng Liu, Rui Wen, Xinlei He, Ahmed Salem, Zhikun Zhang, Michael Backes, Emiliano De Cristofaro, Mario Fritz, and Yang Zhang. {ML-Doctor}: Holistic risk assessment of inference attacks against machine learning models. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4525–4542, 2022.

[22] Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, and Lars Birkedal. Vmsl: A separation logic for mechanised robust safety of virtual machines communicating above ff-a. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1438–1462, 2023.

[23] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Boomerang: Exploiting the semantic gap in trusted execution environments. In *NDSS*, 2017.

[24] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. Darknetz: towards model privacy at the edge using trusted execution environments. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 161–174, 2020.

[25] Myungsuk Moon, Minhee Kim, Joonkyo Jung, and Dokyung Song. Asgard: Protecting on-device deep neural networks with virtualization-based trusted execution environments.

[26] Tushar Nayan, Qiming Guo, Mohammed Al Duniawi, Marcus Botacin, Selcuk Uluagac, and Ruimin Sun. {SoK}: All you need to know about {On-Device}{ML} model extraction-the gap between research and practice. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5233–5250, 2024.

[27] YingTat Ng, Zhe Chen, Haiqing Qiu, and Xuhua Ding. Prism: To fortify widget based user-app data exchanges using android virtualization framework. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '25, page 1567–1581, New York, NY, USA, 2025. Association for Computing Machinery.

[28] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified Hardware/-Software Co-Design for remote attestation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1429–1446, Santa Clara, CA, August 2019. USENIX Association.

[29] Oracle. Java Native Interface. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/, 2024.

[30] Google Pavel Senchanka. Example on-device model personalization with tensorflow lite. https://blog.tensorflow.org/2019/12/example-on-device-model-personalization.html, 2019.

[31] Christopher Pulte, Dhruv C Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. Cn: Verifying systems c code with separation-logic refinement types. *Proceedings of the ACM on Programming Languages*, 7(POPL):1–32, 2023.

[32] Jun Qiu, Xuewu Yang, Huamao Wu, Yajin Zhou, Jinku Li, and Jianfeng Ma. Libcapsule: Complete confinement of third-party libraries in android applications. *IEEE Transactions on Dependable and Secure Computing*, 19(5):2873–2889, 2022.

[33] Davide Quarta, Michele Ianni, Aravind Machiry, Yanick Fratantonio, Eric Gustafson, Davide Balzarotti, Martina Lindorfer, Giovanni Vigna, and Christopher Kruegel. Tarnhelm: Isolated, transparent & confidential execution of arbitrary code in arm's trustzone. In *Proceedings of the 2021 Research on offensive and defensive techniques in the Context of Man At The End (MATE) Attacks*, pages 43–57. 2021.

[34] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *Proceedings of the 38th International Conference on Software Engineering*, pages 923–934, 2016.

[35] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: verification of machine code against authoritative isa semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 825–840, 2022.

[36] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 67–80, 2014.

[37] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *NDSS*, 2017.

[38] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Relaxed virtual memory in armv8-a. In *European Symposium on Programming*, pages 143–173. Springer International Publishing Cham, 2022.

[39] Mohit Singhal, Chen Ling, Pujan Paudel, Poojitha Thota, Nihal Kumarswamy, Gianluca Stringhini, and Shirin Nilizadeh. Sok: Content moderation in social media, from guidelines to enforcement, and research to practice. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 868–895, 2023.

[40] Square. OkHttp3 library. https://github.com/square/okhttp, 2024.

[41] Zhichuang Sun, Ruimin Sun, Changming Liu, Amrita Roy Chowdhury, Long Lu, and Somesh Jha. Shadownet: A secure and efficient on-device model inference system for convolutional neural networks. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1596–1612. IEEE, 2023.

[42] Zhichuang Sun, Ruimin Sun, Long Lu, and Alan Mislove. Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps. In *30th USENIX security symposium (USENIX security 21)*, pages 1955–1972, 2021.

[43] Chia-Che Tsai, Donald E Porter, and Mona Vij. {Graphene-SGX}: A practical library {OS} for unmodified applications on {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.

[44] Hao-Jung Wei, Leng-Kai Lin, Chun-Yen Lin, and Shih-Wei Li. Measuring and optimizing the performance of the android virtualization framework. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, pages 1533–1535, 2024.

[45] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. A first look at deep learning apps on smartphones. In *The World Wide Web Conference*, pages 2125–2136, 2019.

[46] Shunsuke Yoshimura, Kazuaki Nakamura, Naoko Nitta, and Noboru Babaguchi. Model inversion attack against a face recognition system in a black-box setting. In *2021 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pages 1800–1807. IEEE, 2021.

[47] Terence Zhang. Google I/O Event. https://android-developers.googleblog.com/2024/06/top-3-updates-for-building-with-ai-on-android-google-io-24.html, 2024.

[48] Fengmin Zhu, Michael Sammler, Rodolphe Lepigre, Derek Dreyer, and Deepak Garg. Bff: foundational and automated verification of bitfield-manipulating programs. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):1613–1638, 2022.