DEEPVMUNPROTECT: Neural Network-Based Recovery of VM-Protected Android Apps for Semantics-Aware Malware Detection

Xin Zhao, *Member, IEEE*, Mu Zhang, *Member, IEEE*, Xiaopeng Ke, Yu Pan, *Member, IEEE*, Yue Duan, *Member, IEEE*, Sheng Zhong, *Fellow, IEEE*, Fengyuan Xu, *Member, IEEE*

***Abstract*—The emerging virtual machine-based Android packers render existing unpacking techniques ineffective. The state-of-the-art unpacker falls short because it relies on unreliable heuristics and manually crafted semantic models. Hence, it cannot precisely recover app semantics necessary for malware detection. In this paper, we propose DEEPVMUNPROTECT, a deep learning-based approach to automatically and accurately capture the semantics of VM-packed code, so as to facilitate semantic-based Android malware classification. Experiments have shown that DEEPVMUNPROTECT outperforms the state-of-the-art tool on recovering opcode semantics in *Qihoo*(58.3%), *Baidu*(47.5%) and *NMMP* (58.8%) respectively, and can enable semantics-aware malware detection which prior work fails to do.**

***Index Terms*—Android malware; vm-packed application; semantics recovery; deep learning**

I. INTRODUCTION

ANDROID packers can effectively conceal real app content, and therefore have been extensively used by app developers to protect intellectual property. However, malware authors have also exploited this technique to evade detection. A real-world study [1] shows that a large corpus of Android malware has leveraged packing services to hide malicious code. Even worse, a recent report [2] has indicated that threat actors have been utilizing advanced virtual machine (VM)-based packers to encrypt malware.

Prior efforts [3], [4], [5], [1], [6] have thus been made to uncover hidden code from packed Android apps so as to capture malicious *semantics* [7], [8]. However, existing work has largely aimed to precisely identify decrypted DEX code loaded in memory (using heuristics [3], [4], runtime monitoring [5], [1] or hardware assistance [6]), and therefore cannot address the emerging VM-packed apps [2], [9], as the identified memory contents now are no longer Dalvik instructions and cannot be recognized or interpreted without a custom VM embedded in packed apps.

While recovering semantics from VM-based malware has been well studied in traditional desktop systems [10], little has been done to address this unique challenge in the Android setting, where packer VMs are heavily entangled with the Android runtime. To solve this problem, a recent study, **Parema** [9], has proposed to uncover semantics of VM-packed opcodes based upon the essential mapping between original Dalvik bytecode and generated ARM instructions. Unfortunately, despite its effectiveness in revealing the semantic categories of packed bytecode, **Parema** is still fundamentally limited because it relies on *unreliable heuristics* and *manually crafted semantic models*.

In particular, (1) **Parema** exploits the loop structure of the dispatching mechanism in VM interpreters to distinguish VM dispatcher code from translated instructions. However, this heuristics-based approach is not accurate, as the similar loop structures in translated code are prevalent. (2) It manually defines how to reconstruct semantics from machine code (i.e., VEX IR of machine code). Yet it is extremely hard, if not impossible, for a manually created model to precisely capture the nuances in similar instructions from the same "semantic categories", or to comprehensively cover the execution variants of the same code semantics caused by varied inputs and runtime contexts. Consequently, **Parema** cannot serve the needs for semantics-aware detection of VM-packed Android malware, which requires an **accurate** understanding of individual Android opcodes [11], [8], [12], [13], [14], [15], [16].

To address the limitations of the prior work, we propose to combine dynamic analysis with deep learning to automatically and accurately distill the semantics from machine code, so as to facilitate semantic-based malware detection. To correctly identify translated machine code, we resort to the inherent behaviors of VM dispatchers rather than imprecise code structures. To capture instruction semantics in a context-aware fashion, we model bytecode operations using comprehensive ARM instruction traces and Android-specific JNI calls. To precisely recover the correlation between the generated machine code and its original Dalvik bytecode, we invent a custom neural network model, based upon *Bi-LSTM*, *Attention* mechanism and *RNN*, that can systematically distill intrinsic semantics and contexts of individual Dalvik opcodes from their unique machine translations, while tolerating nuances caused by runtime environments and packer implementations.

To the best of our knowledge, we are the first to apply deep learning algorithms to unpacking VM-protected Android apps and thus to enable semantics-aware malware detection in the presence of this novel and advanced evasion technique.

More concretely, we *(a)* first leverage the intrinsic *memory*

X. Zhao, X. Ke, S. Zhong, and F. Xu are with the National Key Lab for Novel Software Technology, Nanjing University, Nanjing, China, 210023. E-mail:{zhao_xin,xiaopeng.ke}@smail.nju.edu.cn, {fengyuan.xu,zhongsheng}@nju.edu.cn

M. Zhang and Y. Pan are with the University of Utah, Salt Lake City, UT 84112, USA. E-mail: {muzhang,yupan97}@cs.utah.edu

Y. Duan is with the Singapore Management University, 81 Victoria St, Singapore 188065. E-mail: yueduan@smu.edu.sg

(Corresponding author: Fengyuan Xu)

access pattern of VM dispatchers to *precisely* identify the translated machine code for each original bytecode instruction. Then, to *comprehensively* capture the semantics of every bytecode operation, we *(b)* formally define the *context-aware opcode semantics*, and collect ARM instruction sequences and JNI call traces to construct the defined semantic model. Eventually, we *(c)* develop a custom deep learning model to *automatically* correlate consecutive ARM instructions with a semantic-equivalent Dalvik opcode. Our model leverages LSTM and RNN to encode instruction and JNI semantics and employs a bidirectional structure plus a self-attention mechanism to capture execution contexts. To train our model, we use a real-world VM-based packer to create 34K labeled bytecode instructions. Once we have recovered opcodes from VM-packed apps, we can therefore enable semantic-based malware detection by feeding opcode sequences to state-of-the-art malware classifiers.

We have implemented DeepVMUnProtect, in 19k lines of Python code, that can automatically recover Dalvik opcode sequences from VM-packed apps. We further use the recovered opcodes for malware detection based upon a state-of-the-art semantic-based Android malware classifier [11]. To evaluate the effectiveness of our tool, we have applied both DeepVMUnProtect and **Parema** to recovering 340 Android apps (benign and malicious), where 80K lines of bytecode instructions have been encrypted using a high-profile VM-based packer. Experimental results have shown that DeepVMUnProtect significantly outperforms the state-of-the-art tool – while 92% of the malware instances recovered by DeepVMUnProtect are successfully detected, only 22% of the malware unpacked by **Parema** can be correctly labeled.

In summary, this paper makes the following contributions:

- We are the first to detect VM-protected Android malware in a semantics-aware manner.
- We are the first to apply machine learning techniques to automatically and accurately recovering semantics from Android apps encrypted by VM-based packers.
- Our prototype DeepVMUnProtect outperforms the state-of-the-art tool on recovering opcode semantics on *Qihoo*(58.3%), *Baidu*(47.5%) and *NMMP (58.8%)*. We have made our tool and our collection of the largest Android VM-packed applications publicly available at **https://github.com/HGWXX-7/DeepVMUnProtect**.

II. Background and Overview

In this section, we introduce Android VMP and discuss the limitations of the state-of-the-art unpacker and our solution.

A. Android VMP Mechanism

Packer Workflow. Figure 1 depicts the high-level overview of an Android VM-based packer. Android executable code is a list of Dalvik bytecode instructions compiled as a `classes.dex` file in an Android app. During execution, the Dalvik Virtual Machine (DVM) interprets these bytecodes. Since the entire DVM runs on an ARM CPU, ARM instructions are generated throughout execution, forming what is known as the execution trace.

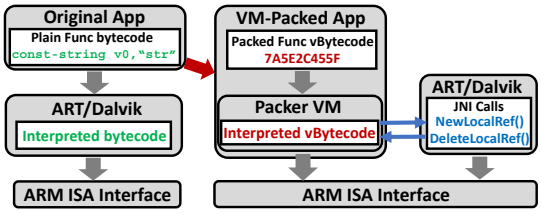


Fig. 1: Virtual Machine-based Packing in Android

To obscure plain Dalvik bytecode, a VM-based packer converts original Dalvik instructions to another binary format (or *vBytecode*), which can only be interpreted by its custom VM. In theory, one Dalvik bytecode could expand to multiple vBytecode; in practice, real-world packers convert one original statement to one encrypted instruction. Note that, however, the way to establish such a one-to-one mapping constantly changes for individual packing processes, because packers commonly use a one-time encoding parameter when encrypting each application instance [9].

Given an Android app, the packer will create a new app package that contains both the VM-packed bytecode (vBytecode) and the unpacker VM. At runtime, a custom VM, rather than the Android interpreter, decodes and executes the vBytecode. To ensure execution efficiency, packer developers implement this VM at the native level instead of the Java level [17]. However, the packer VM cannot handle all operations natively and requires access to the Android runtime for specific data and functions. As a result, the packer VM makes calls via the Java Native Interface (JNI) to interact with the ART or Dalvik VM [17], enabling it to utilize Android framework APIs or object references to perform certain bytecode functionalities.

Packer VM. Figure 2 depicts the inner-working of a packer VM. Upon receiving a packed vBytecode function, the VM first ① fetches a vBytecode from the function, and then ② decodes it to generate a custom instruction that can be processed by the VM interpreter. The decoded vBytecode has two parts: an opcode and operands. The opcode is used by the VM dispatcher to ③ select the corresponding instruction handler, and therefore has major impact on execution semantics; the operands are further fed into the identified handler as a parameter, and may affect how the opcode is interpreted. A selected handler then ④ executes ARM machine code for an original bytecode instruction. The ways to implement these machine instructions, however, can vary greatly depending on whether the corresponding bytecode requires accesses to Android runtime. Some handlers such as `const/16` can be implemented using solely native code. In contrast, other handlers (e.g., `const-string`) must ④A utilize JNI functions to manipulate references in the Android context. Eventually, when execution completes, the VM ⑤ updates the program counter to obtain the next vBytecode.

Our Insight. Our key insight is that *ARM instructions generated by the packer VM eventually reveal original bytecode semantics*, regardless of bytecode-to-vBytecode translation. Therefore, **as a general approach**, to uncover the logic of VM-packed bytecode, one can instead look into the corresponding machine code. This, however, is a challenging task

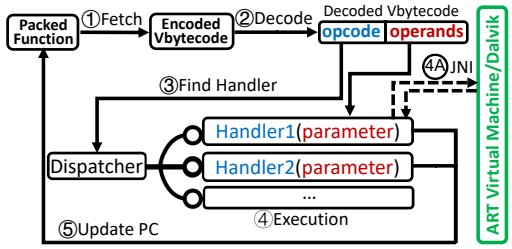


Fig. 2: Execution Flow of Packer VM

due to three technical obstacles: *(1)* how to identify ARM instructions that are associated to individual bytecode; *(2)* how to model the semantics of the identified machine code; and *(3)* how to determine the semantics of constructed models.

B. Limitations of the State of the Art

To address the aforementioned challenges, **Parema**, the only major systematic study on VM-based Android packers, proposed to *(1)* use loop detection to discover the VM dispatcher and thus the machine code in subsequent handlers, *(2)* model the semantics of handler code using manually defined "semantic expressions" – an abstract representation of control-flow logic, and *(3)* reveal opcode semantics via comparing their semantic expressions with predefined models for individual "semantic categories". Yet we argue that these three enabling techniques are fundamentally limited.

a) Imprecise Machine Code Identification.: As a VM dispatcher is executed in a loop and handler code is exercised in individual branches, **Parema** uses loop detection as an approximation to dispatcher (and handler) discovery. Such an over-approximation can lead to significant imprecision, demonstrated in Figure 3.

This figure shows the partial control-flow graph constructed from the ARM execution trace of an example VM-packed app. This bytecode program uses 8 Dalvik opcodes, such as `new-instance`, `invokevirtual/range`, and is packed by *Qihoo 360*, a VM-based packing service. In this case, **Parema** will expect to identify handlers' machine code via first discovering the red loop, which starts from the VM dispatcher and is followed by handler branches. However, merely detecting loops is not sufficient to uniquely and precisely find the dispatcher, as many other loops also exist – to our study, all the brown and blue nodes in the graph involve loops used in handler code. Hence, tedious manual efforts are further needed by the prior work to validate the detected loops [9].

b) Incomplete Semantic Modeling.: To capture the common logic flow of the same handler code in different execution and packing contexts, **Parema** first converts extracted machine code to a symbolic representation to eliminate the impact of varied operands, and then performs a differential analysis to further identify only the constant logic across multiple runtime contexts. This context-agnostic design will unavoidably exclude actual key semantics from their models because even the identical high-level semantics can be implemented fundamentally using different logic flows in individual contexts.

Figure 4 depicts an example of opcode handling, where contexts greatly affects code semantics. More concretely, in this case, `const-string` is executed three times in the original

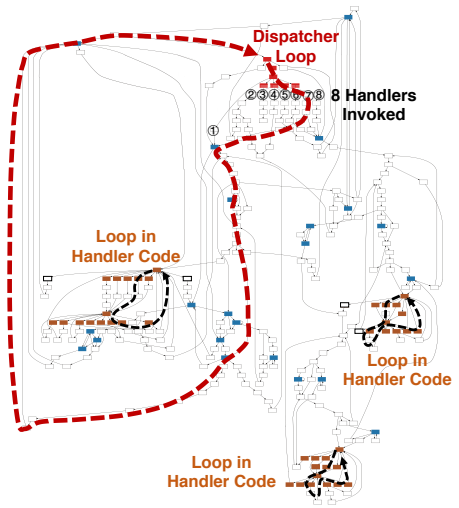


Fig. 3: VM Dispatcher Identification

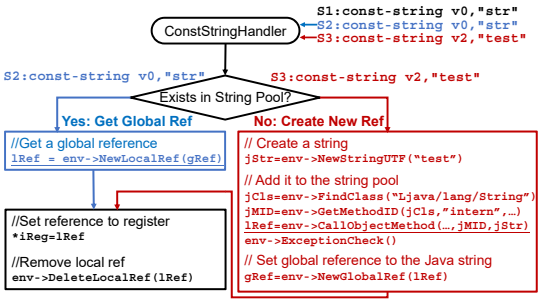


Fig. 4: Different Flows of `const-string` Handler

bytecode. While the first two statements (S1 and S2) store the same string constant "str" to a register, the third (S3) loads another one "test". Because S2 accesses an already existing string reference, the left branch handler will be taken to directly fetch the global reference `gRef` from the string pool, so as to generate a local copy `lRef`. On the contrary, S3 uses a new string, which therefore needs to be created as a string object `jStr` first, and then a local reference `lRef`, and eventually a global reference `gRef`; hence, the right branch is taken. The two branches are implemented using different JNI calls but will merge in the end when the string reference is no longer needed and can be recycled via `DeleteLocalRef(lRef)`.

Since **Parema** focuses solely on the identical portion of multiple handler executions, it will possibly model the semantics of `const-string` using merely the code after the merge point. Yet, although the implementations in the blue and red branches seem different (e.g., `lRef = NewLocalRef(gRef)` vs. `lRef = CallObjectMethod(...,jStr)`), they in fact bear the same high-level semantics – to produce a local string reference. This crucial semantic information, however, is not captured by **Parema**'s semantic models (i.e., semantic expressions).

c) Manually Defined Similarity Metrics.: **Parema** eventually obtains the machine code semantics using a similarity algorithm. It manually defines the similarity – between an extracted semantic expression and a predefined model learned from Android framework code – based upon their Jaccard distance. However, individual features in a semantic model play a different role when they are used to compute a similarity

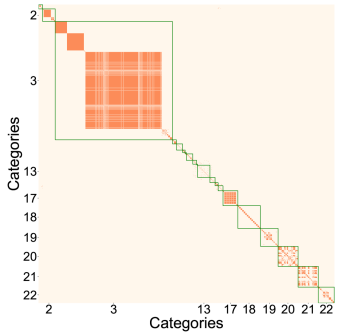


Fig. 5: Pairwise Similarity between Classes in **Parema**

score. Yet such a difference (i.e., weight) cannot be precisely reflected by a manually created metric.

To tolerate such imprecision, **Parema** performs a coarse-grained similarity check – it puts 217 opcodes into 22 manually defined classes, and checks the similarity for each class rather than each opcode. However, manual efforts are not only ad-hoc but also error-prone. To understand this limitation, we follow **Parema**'s approach to generate symbolic expressions for most (211) Dalvik opcodes based on corresponding ARM instructions produced by *Qihoo 360*, and compute the similarity for every pair of opcodes. Figure 5 is a heat map that shows the result. Here, both x-axis and y-axis represent opcodes arranged in the same order and grouped into the 22 classes specified in **Parema**. A darker coloring indicates a higher similarity score. As shown, although the opcodes from one class are generally dissimilar to those in another, the opcodes in the same group are not always similar to each other. For instance, there actually exist multiple major clusters in the Class 3, indicating the inaccurate manual classification.

C. Problem Statement

Having learned a lesson from the existing work, we argue, to effectively extract bytecode-level semantics from translated machine code, we must achieve the following design goals.

Precise. We must precisely locate the binary instructions translated from each bytecode instruction, depending solely on intrinsic behavior of VM dispatchers rather than unreliable heuristics.

Context-Aware. Bytecode semantics must be represented using corresponding machine code in a context-aware fashion. A comprehensive semantic model must be well defined to capture necessary features.

Automatic. When using code similarity to recover semantics, similarity metrics must be systematically derived, and similarity scores must be automatically computed to accurately reflect opcode-level differences.

To this end, we propose DeepVMUnProtect that combines dynamic program analysis with deep learning to automatically and accurately recover opcode semantics(i.e., *invoke-super, invoke-static, const* in Figure 6) from VM-packed Android apps. Figure 6 depicts the whole four steps of our design:

(0) App Filtering. For an unknown packed app, we first preprocess to confirm VM-based encryption. We apply unpackers [1], [6], [3], [5] and check if original DEX can be extracted from memory. If extracted contents lack valid Dalvik

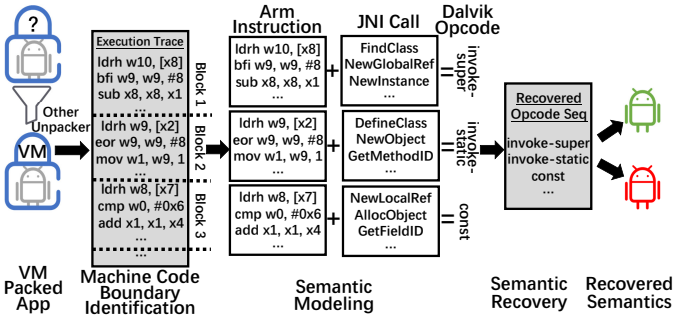


Fig. 6: Overview of DeepVMUnProtect

instructions, we consider the app VMP-encrypted and proceed to the next step.

(1) Machine Code Boundary Identification. Once we have confirmed that an Android app is encrypted by VM-based packers, we run the app using dynamic analysis, and examine its ARM instruction trace, and then identify the boundaries of binary instruction blocks that are associated with individual Dalvik bytecode statements (Section III).

(2) Semantic Modeling. To model the behavior of each binary code block, we formally define the *context-aware opcode semantics*. Our model uses an application-level ARM instruction sequence plus JNI call modeling to represent contextual semantics of each Dalvik opcode (Section IV).

(3) Semantic Recovery. To infer the semantics of constructed models, we train a neural network model to automatically correlate binary instructions to opcodes. To capture the critical yet subtle semantic and contextual features in execution traces, we develop a custom algorithm that combines Bi-LSTM with the self-attention mechanism (Section V).

D. Threat Model

We define our threat model to clarify the goals, capabilities, assumptions, and potential impacts of our work.

- **Goal**: Our goal is to accurately recover the opcode sequence of *VM-packed* Android applications produced by a specific packer manufacturer, aiming to enhance Android malware detection via opcode semantics. Other packing methods are outside the scope of this work.
- **Capabilities**: We have the ability to upload a *limited number* of unpacked applications to the packer manufacturer or collect both unpacked applications and their packed versions. This provides sufficient training data for a deep learning model to capture the opcode semantic patterns specific to the target packer.
- **Assumptions**: The packer manufacturer allows a *limited number* of uploads. One recovery model is trained per packing service, and it can handle different versions of the same packer. This approach is feasible as opcode semantics from a single manufacturer generally follow a consistent distribution, unlike those from different manufacturers.
- **Impacts**: Our work aims to improve the detection of VM-packed Android malware by enabling semantic analysis of obscured opcode sequences. By recovering these patterns, our method enhances the identification of malicious

behaviors concealed within VM-based packing, helping bridge gaps in current malware detection tools.

III. Machine Code Boundary Discovery

A. Algorithm

Identifying the boundary between two machine code blocks, each of which is translated from individual bytecode statements, is equivalent to separating two consecutive translation operations. To do so, we need to depend on the robust patterns of VM interpreters.

Essentially, **Parema** detects the interpretation activities based upon a *code execution pattern*. Because a VM always performs a sequence of actions, *fetch* $\rightsquigarrow$ *decode* $\rightsquigarrow$ *dispatch* $\rightsquigarrow$ *handlerN* $\rightsquigarrow$ *PC++*, in an iterative manner (blue arrows in Figure 7a), a loop pattern must exist. However, this loop-pattern-based approach is fundamentally imprecise, since it only examines how code is executed but does not consider what a loop is used for. In fact, the key differences between a VM interpreter and a normal loop are the types of the data items and memory regions being accessed. Specifically, a VM interpreter uniquely retrieves a new *instruction* to be translated every time, and thus must repeatedly access a code region (red arrows in Figure 7a). Thus, we propose relying on *memory access patterns* to precisely identify packer VM iterations, clearly separating machine code from individual translations.

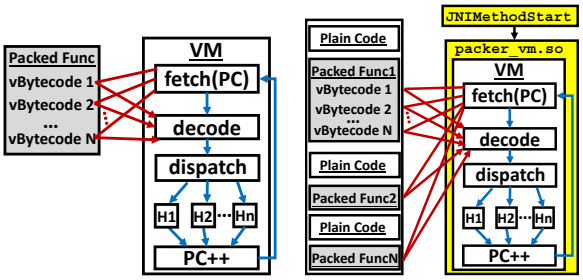
Algorithm 1 conceptually illustrates our basic idea. In general, given a DEX file that has been loaded in memory, we hope to first identify the code regions where packed code is located (Ln.4). Then, we monitor the memory accesses to these regions in the binary execution trace of this DEX program (Ln.6). Because these code items are read by the *fetch* operation in the packer VM, we thus can pinpoint the *fetch* routine from the trace (Ln.7). The invocation of the *fetch* routine indicates the beginning of a code interpretation, and therefore by observing its occurrences, we can correctly separate multiple subsequent translation blocks (Ln.10-12).

Note that, unlike **Parema** which aims to collect only the handler code for semantic extraction, we actually retrieve all the binary instructions between two consecutive calls to the *fetch* routine. Each of our collected translation blocks then involves not only the handler code but also the decoder and dispatcher code. Although this may seem to be less precise compared to the prior work, it does not eventually affect the accuracy of our semantic recovery because our machine learning-based technique can automatically recognize the handler code that varies for different bytecode semantics. In doing so, we actually avoid error-prone manual efforts used in **Parema** to find handler routines.

B. Implementation

Although straightforward, our key idea faces technical challenges in implementing the basic algorithm.

Locating Distributed Packed Code. Ideally, one would expect to obtain a contiguous memory block of packed code. However, in practice, since a DEX file can be partially encrypted, packed code can be divided to multiple portions distributed in an entire DEX file, as shown in Figure 7b. Only



(a) VM Interpretation (b) Real Implementation

Fig. 7: Memory Access (red) vs. Loop Pattern (blue)

Algorithm 1 Translation Blocks Identification

```
1: procedure GetTranslatedBlocks(Trace, DEX)
2:     Blocks ← ∅
3:     FetchRoutines ← ∅
4:     MEM_code ← FindPackedCode(DEX)
5:     for inst ∈ Trace do
6:         if ReadMem(inst, MEM_code) then
7:             FetchRoutines ← FetchRoutines ∪ inst
8:         end if
9:     end for
10:    for f ∈ FetchRoutines do
11:        block ← GetInstBetween(Trace, f, f_next)
12:        Blocks ← Blocks ∪ block
13:    end for
14:    return Blocks
15: end procedure
```

packers' VM has the ability to locate and execute the encrypted code regions, which is untraceable from the data structure of the DEX header. Consequently, to find these encrypted code regions, we first perform a static analysis on the DEX header to collect all the method IDs and code item offsets. Next, we traverse the methods in the DEX file using its header information to identify all the reachable sections. Then, the remaining sections will be the candidates that may contain packed code. To further ensure the existence of packed code in these disconnected sections, we deploy hardware breakpoints to these regions and observe whether they are accessed when a packer VM is running. Note that we intentionally use hardware breakpoints instead of software breakpoints because the latter can be easily detected by many anti-debugging techniques used prevalently in major packers.

In practice, due to Android's limitation of only 4 hardware breakpoints (HWBPs), `DeepVMUnProtect` employs a 'set-hit-remove-reposition' strategy. Specifically, we initially set 4 HWBPs within the code region of the dex file and wait for one of them to be triggered. Once an HWBP is hit, it is removed and repositioned to a subsequent memory address. However, since 4 HWBPs are insufficient to cover all possible code regions, the placement of HWBPs may occasionally fail to intercept the execution of a packed application. In such cases, we reposition the 4 HWBPs to different code regions until they are either triggered or the region is fully covered.

To be conservative, we can add breakpoints to any possible code offsets. However, this is extremely inefficient in reality. To be practical, we also search for the size information that can be possibly obtained at the head of each candidate section. We may then infer that the block of the indicated size belongs to the same logical region, and therefore only insert one

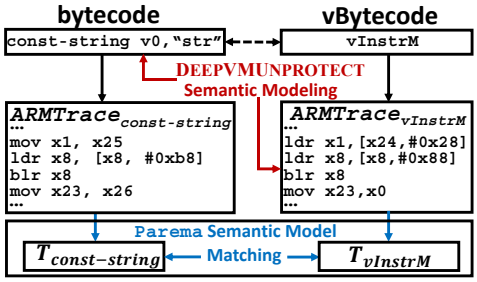


Fig. 8: Semantic Modeling Comparison

breakpoint to the head of an entire block. We understand that there exists a trade-off between exploitation and exploration when using such a heuristic; and, to be safe, we can always fall back to our less efficient approach.

Timing of Monitoring. Another challenge is how to reliably monitor accesses to potential code sections. To exhaustively observe all the memory reads caused by an embedded VM interpreter, we must begin tracing app execution before the VM starts. To address this problem, we depend on our knowledge about the implementations of real-world packers. To our observation, the VM code of existing packers is always compiled into a shared library, and then, at runtime, loaded from a `.so` file and executed through the Java Native Interface (JNI). This is probably because an independent, native implementation is easier to integrate, faster for runtime execution, and harder to reverse engineer. Hence, to guarantee that we can see the start of the VM, we hook the `JNIMethodStart()` function, which is the entry point of any JNI method calls.

Improving Code Coverage. We use dynamic analysis to collect execution traces, and therefore our observation is fundamentally limited by code coverage. While this is a classic challenge for any dynamic analysis-based methods, we devise mechanisms to specifically mitigate this issue in the Android setting. First, we use a forced execution technique and proactively invoke any Java methods discovered from a given app. A forced execution may fail due to the lack of necessary contexts. Hence, we use an existing tool Frida [18] to automatically generate well-formed Android Intents that can enable correct transitions between different contexts.

IV. Formal Semantic Modeling

A. Semantic Equivalence

To understand how to capture original bytecode semantics from machine code, we first formally define semantics for VM translated ARM instructions. Particularly, we follow the approach of the state-of-the-art work on semantic-based malware detection [7], and define semantic equivalence based upon identical changes to *states* caused by executing different instructions or instruction sequences.

Specifically, a *state* s for an instruction or an instruction sequence I is a 3-tuple (val, pc, mem), where $val : Reg \rightarrow Values$ is an assignment of values to the set of registers Reg; pc is a value for the program counter; and $mem : Addr \rightarrow Values$ gives the memory contents. $mem[a]$ denotes the value stored at address a.

If we execute an instruction i from I in state s_0, we transition to a new state s_1 generating an event e. An event can be a "system" event, such as a *libc* call, an Android library function call via JNI, or VM interpretation. If a transition does not generate a system event, $e = null$. Then, we denote a state change from s_0 to s_1 that creates an event e as $s_0 \xrightarrow{e} s_1$.

Definition 1 We say that an ARM instruction sequence I_A exhibits the same behavior of an Android bytecode instruction I_B if there exists an ARM program state s_0^A and a bytecode program state s_0^B, such that $mem(s_0^A) = mem(s_0^B)$ and the state transition sequence $\sigma(I_A, s_0^A)$ generated by executing I_A from the initial state s_0^A is finite, and the follow two conditions hold:

- **Condition 1:** Let the ARM instruction sequence and the bytecode instruction be given as follows:
$$\sigma(I_A, s_0^A) = s_0^A \xrightarrow{e_1^A} s_1^A \xrightarrow{e_2^A} \ldots \xrightarrow{e_p^A} s_p^A$$
$$\sigma(I_B, s_0^B) = s_0^B \xrightarrow{e_1^B} s_1^B$$
Let $affected(\sigma(I_A, s_0^A))$ be the set of addresses a such that $mem(s_0^A)[a] \neq mem(s_p^A)[a]$. Thus, $affected(\sigma(I_A, s_0^A))$ is the set of memory addresses whose value changes after running the ARM instructions from the initial state. We require that $mem(s_p^A)[a] = mem(s_1^B)[a]$ holds for all $a \in affected(\sigma(I_A, s_0^A))$. i.e., values at addresses belonging to $affected(\sigma(I_A, s_0^A))$ are the same after executing the ARM instruction sequence I_A and the bytecode instruction I_B.
- **Condition 2:** If $pc(s_p^A) \in affected(\sigma(I_A, s_0^A))$, then $pc(s_1^B) \in affected(\sigma(I_A, s_0^A))$. i.e., if the program counter at the end of executing I_A points to the affected memory area, then the program counter after executing I_B should also point to the affected memory region.

B. Context-Aware Opcode Semantics

The formal semantics in **Definition 1** can be overly strict [7], since not all the memory changes have permanent impacts. If memory locations are used to store temporaries, they should not be checked for semantic equivalence. Hence, we can define core memory areas that must be examined when comparing the execution results of a binary instruction sequence I_A and a Dalvik bytecode I_B.

There could be multiple ways to define the core memory regions. One suggested option [7] is to label each instruction as either `temp` or `persistent`, and only investigate memory accesses performed by `persistent` instructions. Nonetheless, this coarse-grained labeling cannot be applied to I_B which has only one single bytecode instruction. Thus, we further distinguish individual parts in one instruction using two labels `constant` and `variable`. In particular, we consider a Dalvik *opcode* can make a constant behavioral impact. In contrast, the impact of *operand* values is temporary and indefinite – operand values can vary from time to time due to compilation (e.g., register assignments); individual values may or may not cause unique consequences. For example, in the case of `const-string v2,''test''`, the opcode can always introduce a new local string object. In comparison, the first operand can be assigned to another virtual register (memory region) in a different compiled version; two string instances for the second operand – regardless of their concrete values – can possibly lead to the same or different logic flows (depending rather on whether they are present in the constant pool).

We therefore define a **weaker semantics** based upon the association between an Android *opcode* op_B and an ARM instruction sequence I_A. Though less precise, opcode-based semantics have proven effective in previous Android malware detection [11], [12].

Also note that, **Parema** uses opcodes to model the behavior of translated machine code as well. Though **Parema** can recover operand data, it does not leverage it to recover code semantics. In fact, it depends solely on opcode semantics to interpret the logic of corresponding ARM instructions. Besides, **Parema**'s operand recovery relies on an overly strong and thus less realistic assumption that original bytecode and packed vBytecode share the same operand values and encoding.

Nevertheless, when modeling opcode semantics, **Parema** does not consider the execution context of opcode. However, as shown in Section II-B, the behaviors of the same opcode can be drastically different in distinctive contexts. Consequently, we argue that although concrete operand values are not evaluated in opcode semantics, the changes to execution context caused by operands must be taken into account.

Figure 8 depicts the difference between the prior work and our modeling approach. In particular, **Parema** uses a *template*-based semantic model [7], which seeks to find the equivalence between a semantic template T_{op_B} for an Android opcode op_B (e.g., `const-string`) and a template T_A for an ARM instruction block I_A (e.g., `vInstrM`). Formally, a template T is a 3-tuple (I_T, V_T, C_T), where I_T is an instruction sequence, V_T and C_T are set of variables and symbolic constants in the instructions. An *execution context* EC_T of a template T is an assignment of concrete values to the symbolic constants in C_T. Then, we say an instruction sequence I contains a behavior specified by a template T, if there exists a program state s_0, an execution context EC_T, and a template state s_0^T such that $mem(s_0^T) = mem(s_0)$, and the final state equivalence conditions in **Definition 1** should also hold for the two finite execution sequences $\sigma(I, s_0)$ and $\sigma(T, EC_T, s_0^T) = s_0^T \xrightarrow{e_1^T} s_1^T \xrightarrow{e_2^T} ... \xrightarrow{e_r^T} s_r^T$. *Hence, the semantic equality between a template and an instruction sequence is defined in a context-aware fashion.*

To create these templates, **Parema** uses the packer to produce ARM blocks for known and unknown bytecodes, and thus can generate $T_{const-string}$ and $T_{vInstrM}$, respectively. However, when **Parema** checks the semantic equivalence between `const-string` and `vInstrM`, it simply performs a context-insensitive comparison between the two static templates without the consideration of $EC_{T_{const-string}}$ and $EC_{T_{vInstrM}}$.

In contrast, DeepVMUnProtect preserves the impact of execution context when correlating `const-string` and `vInstrM`. To avoid losing contextual information in intermediate templates, we directly associate an opcode with an ARM instruction block. Therefore, we modify the **Condition 1** in Definition 1 to define our *context-aware opcode-based semantics*:

Modified Condition 1: We require $mem(s_p^A)[a] = mem(s_1^B)[a]$ holds for all $a \in affected(\sigma(op_B, EC_B, s_0^B))$, where $\sigma(op_B, EC_B, s_0^B)$ denotes the execution of op_B starting from a state s_0^B in a context EC_B. i.e., values at addresses that belong to the set $affected(\sigma(op_B, EC_B, s_0^B))$ are the same after executing I_A and I_B.

```
1 ...                          9  ldr    x8, [x8, #0x348]
2 csel   x1, x11, x10, eq     10  blr    x8
3 blr    x9                   11  NewGlobalRef
4 FindClass                   12  ldr    x24, [x28]
5 mov    x23, x0              13  mov    x19, x0
6 cbnz   x0, 0x792154cc1c     14  cbnz   x24, 0x792154cc64
7 adrp   x25, 178             15  str    x28, [sp, #0x8]
8 ...                         16  ...
```

Fig. 9: `invoke-super` Instruction Trace w/ JNI Modeled

C. Modeling JNI Calls

We collect ARM traces to recover bytecode semantics based on our model. In theory, such traces cover the needed information. But in practice, naïve tracing yields excessively detailed traces, including external call instructions. For instance, `invoke-super` generates 11K instructions, 90% from external functions. The large external call volume can dominate traces, deteriorating semantic extraction from the remaining code.

To address this, we model instead of recording external calls. Specifically, we replace call instruction sequences with function names. This models calls context-insensitively. However, system and library calls are usually stateless, so the approximation does not significantly reduce model precision.

Formally, let E_{call}^A be the set of system events (i.e., events that make external calls) in all the events E^A of instructions sequence I_A, then $E_{call}^A = \{e | e \in E^A \wedge e \neq null\}$. If a state $s_{call_n^0}^A$ is triggered by any event in E_{call}^A, then the execution of an external function starting at $s_{call_n^0}^A$ is a state transition:

$$\sigma(I_A, s_{call_n^0}^A) = s_{call_n^0}^A \xrightarrow{e^A_{call_n^1}} s_{call_n^1}^A \xrightarrow{e^A_{call_n^2}} ... \xrightarrow{e^A_{call_n^t}} s_{call_n^t}^A$$

Then, when external calls are modeled, the affected memory areas $affected_{\mathtt{M}}(\sigma(op_B, EC_B, s_0^B))$ is derived as:

$$affected_{\mathtt{M}}(\sigma(op_B, EC_B, s_0^B)) = affected(\sigma(op_B, EC_B, s_0^B)) - \bigcup_n affected(\sigma(I_A, s_{call_n^0}^A)) + \bigcup_n affected(\mathtt{M}_{call_n})$$

where $affected(\mathtt{M}_{call_n})$ models the difference in memory contents between input and output states for an external call.

Thus, the Condition 1 can be further modified in practice:

Further Modified Condition 1: We require that for all $a \in affected_{\mathtt{M}}(\sigma(op_B, EC_B, s_0^B))$, $mem(s_p^A)[a] = mem(s_1^B)[a]$ holds, where $(\sigma(op_B, EC_B, s_0^B))$ denotes the execution of op_B starting from a bytecode state s_0^B in a bytecode context EC_B. i.e., values at addresses that belong to the set $affected_{\mathtt{M}}(\sigma(op_B, EC_B, s_0^B))$ are the same after executing the ARM instruction sequence I_A and the bytecode I_B.

Note that, while we model all the external functions including system calls, C library functions and Android framework methods, we pay special attention to the Android framework functions (i.e., JNI calls) as they bear stronger Android-specific semantics. Android framework functions locate in system native library libart.so, thus all of them can be modeled ahead of time. Figure 9 exemplifies a collected trace corresponding to `invoke-super`, where JNIs (and other external calls) have been modeled. Our trace preserves the occurrence, ordering and context of key JNI calls, while hiding less important JNI internal implementation details.

D. Implementation

In practice, when a hardware breakpoint is triggered, we initiate the tracing of ARM instructions and JNI calls using a customized LLDB or GDB plugin. The specific steps are as follows:

- *Breakpoint Hit.* When a preset hardware breakpoint is triggered, the LLDB or GDB plugin removes the hit breakpoint.
- *Set New Breakpoints.* After the hit address, four consecutive hardware breakpoints are set to capture subsequent instruction fetches (i.e., "vm fetch").
- *Control Flow Analysis.* The plugin uses the LLDB or GDB API to identify the first control flow-changing instruction (e.g., jump) after the current program counter.
- *Handle Control Flow and Functions.* A regular breakpoint is set at the identified instruction. If the instruction is a branch into a `libart.so` function, the function name is recorded, and the "Step Out" command is used to return to the calling address.
- *Iterative Tracing.* The tracing process repeats until the hardware breakpoints are triggered again.
- *Instruction Logging.* All executed instructions are logged and saved to a local file after tracing concludes.

V. Semantic Recovery

Once we have extracted an ARM instruction trace with external calls being modeled, we will recover its semantics via associating it to a specific Android opcode. To establish this correlation, we resort to deep learning techniques. Specifically, we hope to train a neural network model that can automatically label a block of ARM instructions as a Dalvik opcode that shares the similar behavior.

A. Basic Algorithm: Sequence-to-Vector

Deep learning has been proven effective for capturing nuanced similarity and dissimilarity in binary code. For instance, previous research has used RNN [19], [20], long short-term memory model (LSTM) [21] to precisely model the internal dependencies in an instruction trace. Essentially, we can also use LSTM to model our ARM instruction traces, as LSTM can address the derivative vanishing and explosion problems and thus remember information for long periods of time when scanning lengthy instruction traces (e.g., thousands lines of code). Figure 10 illustrates our basic LSTM structure that processes a sequence of instructions. Specifically, we first convert each "token" in an instruction, such as "`ret`" or "`x8`", to a vector representation using `word2vec` [22] and then pass this generated sequence of vectors as inputs to a LSTM. Eventually, we will transform the LSTM outputs to a feature vector, predicting the Dalvik opcode matching the ARM instruction sequence.

A LSTM can handle our input sequence of word embeddings $x^{(1)}, ..., x^{(t)}$ using a series of LSTM cells, and map it to an output sequence $h^{(1)}, ..., h^{(t)}$. Each cell has a state unit $s^{(t)}$ and three gating units: a forget gate unit $f^{(t)}$, an external input gate $e^{(t)}$, and an output gate $o^{(t)}$.

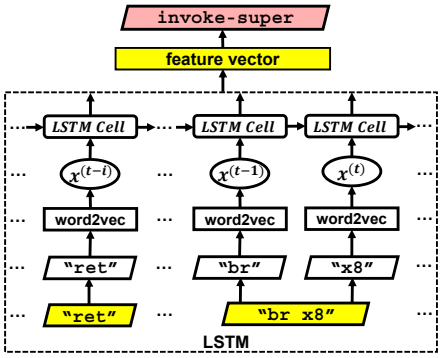


Fig. 10: LSTM Structure for Instruction Trace

B. Instruction and JNI Semantics

a) Instruction-level Semantics.: When encoding instructions into vector representations, it is critical to preserve their semantics. The `DEEPVSA` work [21] indicates that simply modeling an entire instruction trace as one sequence of bytes does not necessarily capture instruction-level semantics. To address this problem, the prior work has proposed a hierarchical model where each instruction is encoded as a byte sequence individually. For instance, an x86 instruction `call[eax]` can be converted into one embedding based on its three bytes `0x67`, `0xff` and `0x10`. Then, all the instruction-level embeddings will be used as inputs for another level of encoding.

To make the architecture of our model simple while still being able to preserve instruction semantics, we take a different approach. In general, we first disassemble an ARM binary instruction trace to generate text-formed disassembly – converting each instruction to opcode and operand string tokens – and then use this sequence of string tokens to produce vector representations. For example, a jump instruction `D61F 0100` (`br x8`) is represented as two string tokens "`br`" and "`x8`". In doing so, we can actually differentiate opcodes and operands that share the same byte values.

However, a simply disassembled string sequence cannot be directly used to generate semantics-aware encoding. Two special challenges must be addressed. First, instructions of the same semantics may seem very different because they take distinctive operands. For instance, each jump instruction such as `bl 0x79214c95f0` has a specific jump target. To indicate their common semantics, concrete operand values must be normalized in advance. To solve this problem, we follow the approach of the prior work [7] and replace concrete parameters with semantic-level string tokens such as "`ADDR`" for jump targets or "`NUM`" for general values.

Second, a disassembled operand string may contain multiple semantic-level components. For instance, in a load instruction `ldr x22, [sp]`, the second operand in fact consists of both a register `sp` and a dereferencing operator `[ ]`. However, using a string token as an entirety (e.g., "`[sp]`") to represent the combination of several semantic elements can lead to an excessive amount of "compound" semantics, and thus bring challenges to both encoding and learning. To tackle this problem, we further separate a single operand string into finer-grained tokens, each of which can be interpreted individually. For instance, in Figure 11, we divide "`[sp]`" into three elements "`[`", "`sp`"

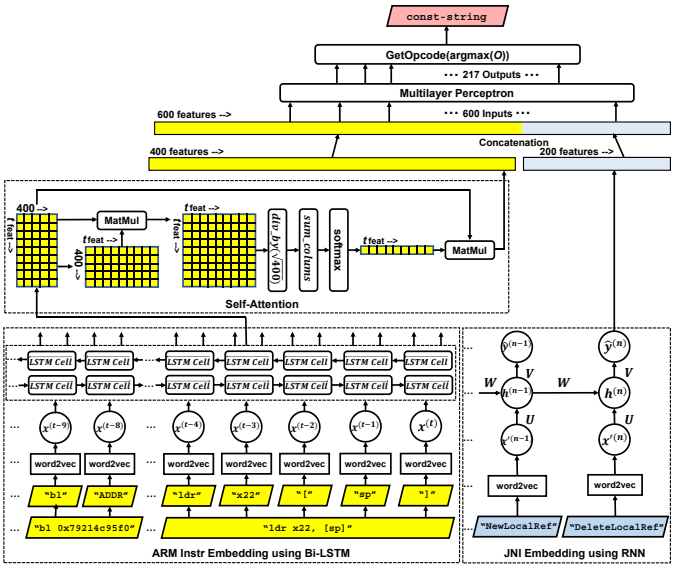


Fig. 11: Semantics and Context-Aware Model

and "]", which clearly indicate the usage of a specific register and how it is used. In this way, we can easily understand the semantic-level resemblance between, for example, `ldr x22, [sp]` and `ldr x28, [sp, #0x30]` – since both load data from an address based on the `sp` register.

JNI Semantics. JNIs play an important role in interpreting instruction behaviors, so we model them separately, in addition to modeling ARM instruction traces, to emphasize their significance. Particularly, we expect to model not only the occurrence of JNI functions but also their temporal order, since the ordering reflects the control and data dependencies in multiple JNIs for conducting specific activities. For instance, to create a new constant string, a sequence of JNIs must be called in this exact order: `NewStringUTF()` ⇝ `FindClass()` ⇝ `GetMethodID()` ⇝ `CallObjectMethod()` ⇝ `ExceptionCheck()` ⇝ `NewGlobalRef()` (Figure 4).

To this end, we extract a JNI call sequence from an ARM instruction trace, and employ a vanilla RNN to model this JNI sequence. Since JNI call sequences are much shorter by nature, compared to instruction sequences, RNNs can already handle them effectively without causing derivative vanishing and explosion problems.

Our RNN structure is depicted on the right side of Figure 11. More concretely, we first retrieve the JNI name (e.g., "`NewLocalRef`") from each JNI call in a sequence, and transform every name string to a vector using `word2vec`, and then send the derived vector sequence as inputs to a RNN model. Similar to a LSTM model, a RNN also processes a sequence of inputs $x'^{(1)}, ..., x'^{(n)}$ and can be trained to map the input sequence to a chain of hidden states $h^{(1)}, ..., h^{(n)}$. To make prediction using this chain structure, a RNN performs a forward propagation starting from an initial state $h^{(0)}$. It then uses the following update equations to predict $\hat{y}^{(n)}$.

C. Execution Contexts

Additionally, in order to recover context-aware semantics (defined in Section IV), our machine learning model must also capture the execution contexts of instruction traces.

Context Extraction via Bidirectional Analysis. While a LSTM captures "forward" dependencies (e.g., temporal order) among individual instructions within an execution trace, it does not model the execution contexts which actually requires *backward* analysis. Consider, again, the `const-string` example described in Section II-B (Figure 4). When a LSTM model processes the instruction sequence forwardly, it first encounters the condition statement that checks whether a string constant exists in the string pool. However, at this point, the model cannot answer this question due to the lack of contextual information about the constant pool's state. In fact, one cannot reason about this earlier context until it reaches a later state in either of the two conditional branches: if a new string is created in the branch being taken, the string under inspection was absent in the previous context; otherwise, the string was present. However, the standard LSTM structure does not look back to recover previous contexts at a later stage. Thus, to be able to observe such a later state earlier, we need to further process an instruction trace in a reverse order. To this end, we additionally adopt a backward LSTM structure, implementing a bidirectional architecture.

It is noteworthy that `DEEPVSA` [21] has also used Bi-LSTM to handle instruction sequences. However, the previous work leverages backward analysis largely to retrieve *data dependencies* between a current and a future instruction – for instance, it can infer the type of a register based on how it is used in the future. In contrast, we hope to capture *control dependencies* between "now" and "future", and use "future" information to determine the concrete runtime contexts at "present" that can lead to the exact future state.

Consequently, as depicted in Figure 11, every input of a word embedding, $x^{(t_i)}$, is now processed by both a forward LSTM structure and a backward one, and thus is translated to two output vectors. Each of these vectors has 200 features.

Self-Attention. Not all the future instructions are of the same importance in terms of inferring a prior context. In Figure 4, when a string constant does not exist in a prior context, the instructions to invoke `NewGlobalRef()`, which creates a new global reference for this string, become a strong indicator of such contextual information. In contrast, other transitional instructions such as calls to `FindClass()` or `GetMethodID()` play a less important role, as they are only "helper" code to support the creation of new objects. To strengthen the correlation between execution contexts and those instructions that actually reflect the contextual impacts, we use *self-attention* mechanism to emphasize more important causal relations.

Specifically, we use the output of Bi-LSTM to calculate the attention weights W_{att}. Concretely speaking, we first concatenate the two output vectors for each Bi-LSTM input $x^{(t_i)}$, to create a 400-dimension feature vector. Then, the input of our self-attention computation is a (t×400) matrix Q, where t is the number of input items in $x^{(1)}, x^{(2)}, ..., x^{(t)}$. Thus, our attention weights W_{att} is derived using the following equation:

$$W_{att} = \textbf{softmax}(\sum_{i=1}^{t}(\frac{QQ^{\top}}{\sqrt{d_Q}})_i)$$

where d_Q denotes the row count of matrix Q; $(\cdot)_i$ indicates the i^{th} row of the matrix; and the **softmax** classifier is used for the normalization purpose. Hence, our instruction embedding network consists of a Bi-LSTM model and a self-attention module (left part of Figure 11). The output feature of an instruction sequence thus is a 400-dimension vector:

$$\begin{aligned} feature_{instr} &= \varphi_{\text{Bi-LSTM+Attention}}([x_1, x_2, ..., x_t]) \\ &= W_{att} \times \varphi_{\text{Bi-LSTM}}([x_1, x_2, ..., x_t]) \\ &= W_{att} \times Q \end{aligned}$$

where $\varphi_{\text{Bi-LSTM}}$ is our Bi-LSTM structure; $\varphi_{\text{Bi-LSTM+Attention}}$ denotes our Bi-LSTM model plus self-attention mechanism.

D. Dalvik Opcode Prediction

To predict the Dalvik opcode corresponding to a given instruction trace, we finally combine both ARM instruction and JNI features via concatenating the outputs of two models, and then feed the combined feature vector into a *multilayer perceptron* (MLP) network. The top part of Figure 11 briefly illustrates this prediction model. In particular, our MLP structure takes a 600-feature vector as an input I, processes the input using one hidden layer H, and generates a 217-dimension output vector O. Each dimension matches a Dalvik opcode, with the highest value as the prediction.

Our hidden layer has 64 neurons. The mapping follows:

$$\begin{aligned} H &= \mathbf{ReLU}(I \times \mathbf{W}' + \mathbf{B}') \\ O &= H \times \mathbf{W}'' + \mathbf{B}'' \\ opcode &= \mathbf{GetOpcode}(\mathbf{argmax}(O)) \end{aligned}$$

where $\mathbf{W}'$ and $\mathbf{W}''$ denote weight parameters: $\mathbf{W}'$ is a 600×64 matrix and $\mathbf{W}''$ is a 64×217 matrix; $\mathbf{B}'$ and $\mathbf{B}''$ represent biases: $\mathbf{B}'$ is a 64-dimension vector while $\mathbf{B}''$ is a 217-dimension vector; **argmax** finds the maximum dimension in the output vector O, and **GetOpcode** maps it to a specific opcode name.

VI. Evaluation

We have implemented DeepVMUnProtect in 23K lines of Python and JS code. Our dynamic analysis is built atop Frida [18], LLDB [23] and GDB. We use PyTorch [24] to implement our ML algorithms.

To assess the usefulness of DeepVMUnProtect, we evaluate the effectiveness of our ML models, the accuracy of opcode recovery, and more importantly, how recovered opcode sequences facilitate semantic-based malware detection. Experiments are conducted on a server equipped with Intel Xeon Gold 6330 CPU @ 2.00GHz, Nvidia TITAN Xp GPU, and 256GB memory. The OS is Ubuntu 20.04 LTS (64bit). Given that some packed applications cannot run on newer versions of the Android system, our experiments were conducted on Android 8 to achieve better compatibility. However, our tool is also capable of running on the latest Android systems, such as Android 13. We will also provide code compatible with Android 13 in our repository [1].

[1]https://github.com/HGWXX-7/DeepVMUnProtect

TABLE I: Datasets for Training/Testing Our Model

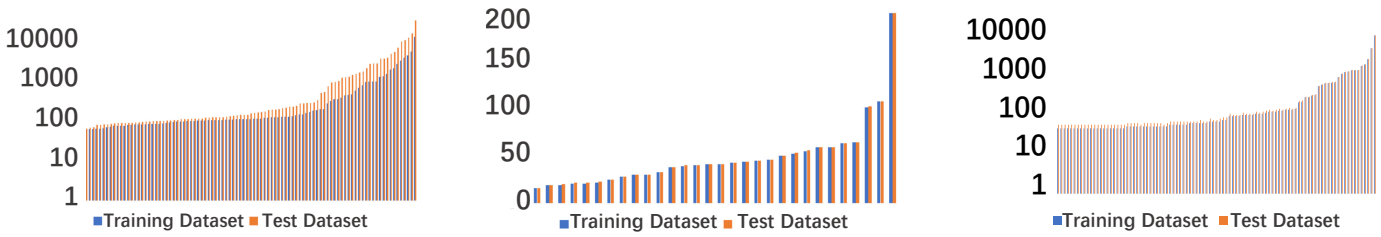
Source	Training	Testing	Total
Qihoo	36,236 instrs	86,180 instrs	122,416 instrs
Baidu	1,429 instrs	1,421 instrs	2,850 instrs
NMMP	55,705 instrs	60,432 instrs	116,137 instrs

A. Data Collection

Data for Training/Testing Our Opcode Recovery Model. To train and test our DeepVMUnProtect model for each packer, we must collect both original and packed bytecode instructions so as to obtain a large volume of ARM instructions labeled with their corresponding Dalvik opcode. In principle, we can achieve this via submitting plain apps to online packing services. However, it turns out to be an extremely challenging task. In practice, very few VM-based packing services are accessible to large-scale studies. Most major packers, such as Baidu packer [25], ijiami [26], Bangcle [27], **charge 3,000–10,000 US dollars per year for packing merely one app instance**. We contacted these packer vendors requesting their assistance but received no response.

Dataset Details. Despite this difficulty, we still manage to establish our dataset, 122,416 packed bytecode instructions from 345 apps, using multiple variations (V3.2.2.3 – V3.5.0.0) of a VM-based packer *Qihoo 360* – the only public service that provides free VM-based packing. In addition, we also collect 2,850 packed instructions, from 20 apps used in the `Parema` [9] project, which have been encrypted by another VM-based packer from *Baidu*. We hope to use the experiments on the *Baidu* packer to demonstrate the **generality** of our approach, as we do not have prior knowledge of how at a low level *Baidu* packer encrypts apps. Moreover, we collected an open-source VM-based packer named *NMMP* on Github [28] with 792 stars and packed 116,137 instructions in 177 applications. Note that the state-of-the-art work [9] has also been evaluated using apps processed by exactly two packers (Baidu and Qihoo). Namely, our evaluation is conducted on the largest VM-packed applications, we will also make our dataset publicly access to facilitate the research on this field. Table I shows the dataset details. However, varying opcode amounts and small quantities of certain training samples can affect model prediction accuracy. Figure 12 depicts the distribution of opcodes (sorted) in both datasets, and we do observe limited numbers of specific opcodes (e.g., `new-instance`, `move-result`, etc.) in *Baidu* dataset.

Representativeness. In fact, these two packing tools dominate the packer market. We have studied 29,269 top Android apps from major markets such as Anzhi [29], Baidu [30], Xiaomi [31], 360 [32] with a rule-based open-source packer identifier [33]. While 48% of these apps are encrypted by packers, 83% of these packed apps are protected by VM-based packers and **77% of the VM-packed apps (8,558)** are encrypted by *Qihoo* or *Baidu*. Such a dominance may result from their high accessibility and zero user background check. Thus, malware authors have already started to use the *Qihoo* service to pack malicious Android apps [2]. In contrast, to our observation, the remaining 23% apps which use premium-rate packing services are often from major and trustworthy organizations (e.g., Alipay, WeChat, China Telecom, Industrial and Commercial Bank of China, etc.). As a result, if we can



(a) **In Qihoo Dataset** (b) **In Baidu Dataset** (c) **In NMMP Dataset**

Fig. 12: Opcode distribution in different datasets. X-axis is the opcode; Y-axis is the occurrence of each opcode.

successfully recover code packed by *Qihoo* and *Baidu*, we can already address a large majority of encrypted and less trustworthy apps in the Android ecosystem and can therefore make a significant impact on Android application security.

Label Generation. An individual label is associated to a block of ARM instructions and indicates which Dalvik opcode these instructions represent. To attach these labels, we rely on our insights into the real-world implementation of VM-based packers – that is, existing packers convert each Dalvik bytecode statement to exactly one single vBytecode instruction. Note that, the state-of-the-art work **Parema** also shares the same observation and builds their solution based upon this key assumption. As a result, we instrument every packed vBytecode to identify an individual ARM instruction block that corresponds to the original bytecode instruction.

Data for Training/Testing Malware Classifier. To test how well DeepVMUnProtect can recovery malware semantics, we train an opcode-sequence-based malware classifier [11] using 5,148 malware samples and 5,606 benign Android apps. Specifically, we collect benign samples from a major app market [32] and malware from the UNB repository [34].

To obtain testing samples, we collect VM-packed apps (malicious and benign) via submitting original plain apps to the *Qihoo 360* packing service. It is non-trivial to pack malicious apps via online service, as it actually performs malware detection and rejects those that do not pass their tests. As a result, it is extremely hard to use well-studied malware samples from classic repositories such as Android Malware Genome Project [35] for our evaluation. To address this problem, we retrieve newer malware instances (i.e., discovered in 2019 and later) from an up-to-date malware repo (UNB repository) [34], and submit all these samples to the packing service. Although still a large portion of these apps is flagged as malicious, we finally manage to obtain 283 malware instances and 57 benign applications that can be packed.

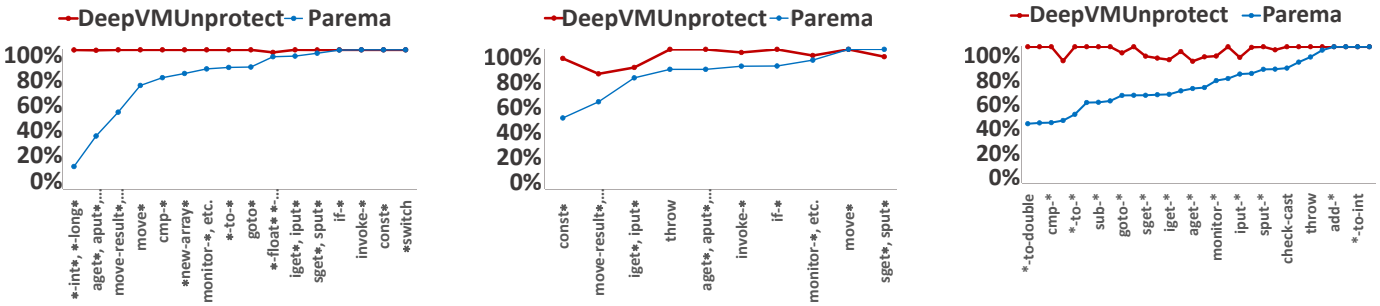
B. Accuracy of Opcode Recovery

We evaluate whether DeepVMUnProtect can correctly recover Dalvik opcode from binary execution traces. Particularly, we would like to compare our system with the state-of-the-art tool **Parema**. To this end, we apply both DeepVMUnProtect and **Parema** to 86,180 bytecode instructions packed by *Qihoo 360*, 1,421 instructions from **Parema** encrypted by *Baidu* and 55,705 instructions packed by *NMMP* . Because **Parema**'s source code has not been fully released, we consulted their authors to reproduce **Parema**'s core algorithms for building and comparing their "semantic expressions".

Category-Level Recovery Accuracy. To enable a head-to-head comparison, we first follow **Parema**'s approach and assess the accuracy of opcode recovery at the "category" level. More concretely, the authors of **Parema** manually classify Dalvik opcodes into 22 categories based upon their relevance. For instance, all the `move` instructions such as `move/from16`, `move-wide`, `move-object` are put into the same category. Due to this coarse-grained setting, manually defined similarity metrics in the prior work can still lead to reasonably accurate results. In general, while DeepVMUnProtect can achieve a better recovery accuracy – 99.9% for *Qihoo*, 93.6% for *Baidu* dataset and 97.9% for *NMMP* dataset, **Parema** can also correctly recover the category of 84.4%, 69.3% and 72.5% instructions, respectively.

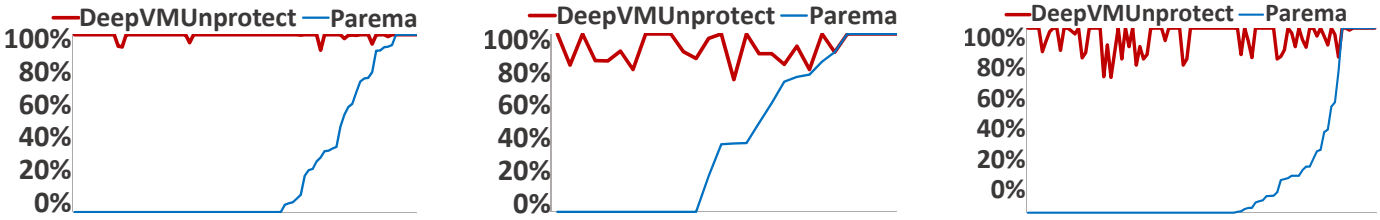
Nevertheless, **Parema** performs very poorly for certain categories. Figure 13 illustrates the category-level recovery accuracy for *Qihoo*, *Baidu* and *NMMP* packers, respectively – note that not all opcodes have been used in these apps, and individual datasets contain different opcodes. While DeepVMUnProtect performs uniformly well for all opcode categories, **Parema** fails to recover most of the opcodes for `*-int*`, `*-long*`, `aget*`, `aput*`, `const*`, `move-result*`, etc. To our study, such inaccuracy is due to two reasons. Firstly, the handlers for certain opcodes (e.g., array access) contain overly complex logic, and thus cannot be precisely summarized as semantic expressions using solely manual efforts. Secondly, an arithmetic operation (e.g., `mul-int`, `add-long`, etc.) is generally translated into a few very common instructions – such a simple pattern is not unique and exists in many other opcode handlers (e.g., `const*`). Hence, although simple, arithmetic opcodes still cannot be well recovered.

Opcode-Level Recovery Accuracy. We then evaluate the recovery accuracy at the opcode level. Figure 14 illustrates the results. While our DeepVMUnProtect can still accurately recover each opcode for *Qihoo* (99.9%), *Baidu* (89.5%) and *NMMP* (95.6%) packed apps, **Parema** leads to a low recovery rate (41.6%, 42%, 36.8%). In fact, **Parema** cannot identify a large amount of Dalvik instructions. For instance, **Parema** cannot precisely differentiate individual `move-*` instructions (e.g., `move`, `move/from16`, `move-wide`) or `const-*` variants (e.g., `const/4`, `const/16`, `const/high16`, `const/wide`) and therefore causes a considerable inaccuracy. This, again, shows that manually defined semantic models and similarity metrics cannot sufficiently capture the nuances of opcodes. In contrast, our machine learning model can precisely quantify subtle differences in their execution contexts to recover opcode semantics.



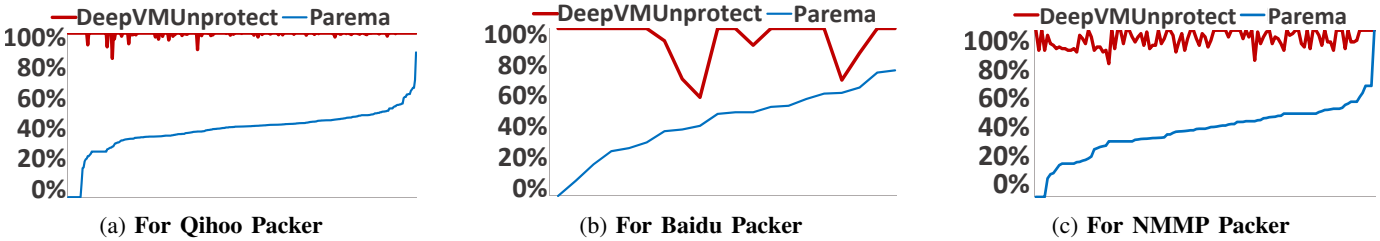
(a) **For Qihoo Packer** (b) **For Baidu Packer** (c) **For NMMP Packer**

Fig. 13: Recovery Accuracy at Category-Level. X-axis is the category; Y-axis is the percentage of recovered instrs.



(a) **For Qihoo Packer** (b) **For Baidu Packer** (c) **For NMMP Packer**

Fig. 14: Recovery Accuracy at Opcode-Level. X-axis is the opcode ID; Y-axis is the percentage of recovered instructions.



(a) **For Qihoo Packer** (b) **For Baidu Packer** (c) **For NMMP Packer**

Fig. 15: App Recovery Accuracy at Opcode-Level. X-axis is the app IDs; Y-axis is the percentage of recovered instrs.

TABLE II: Classification Accuracy

	Malware	Benign	All
Baseline Packed Apps	0/254 (**0%**)	43/43 (100.0%)	43/297 (**14.5%**)
Parema-Recovered	55/254 (**21.7%**)	33/43 (76.7%)	88/297 (**29.6%**)
DeepVMUnProtect-Recovered	234/254 (**92.1%**)	40/43 (93.0%)	269/297 (**92.3%**)

In addition, we also measure how well DeepVMUnProtect and **Parema** can recover opcodes for each application, as malware detection is performed at the application level. As depicted in Figure 15, where x-axes are sorted by accuracy, for 98.5% apps packed by *Qihoo 360*, we can correctly identify more than 95% of the instruction opcodes; for 80% apps encrypted by *Baidu*, we can recover over 90% of their opcodes; for 93% apps packed by *NMMP*, we can correctly recover over 90% of the instruction opcodes. In contrast, **Parema** is unable to recover over 90% of the instructions for any of the apps in either dataset.

Discussion on the results for the *Baidu* dataset. In general, our opcode recovery accuracy for the *Baidu*-packed apps – although much higher than **Parema**'s – is relatively lower compared to our recovery accuracy for the *Qihoo and NMMP* packed apps. This is actually because certain opcodes in the *Baidu* samples are very few, as depicted in Figure 12, and thus are not sufficient for training a very accurate model. In fact, if we only consider those opcodes whose occurrences are higher than 29, we can correctly recover 98.08% of instructions and accurately recover 90.47% of the apps.

C. Semantics-Aware Malware Detection

Next, we assess the effectiveness of DeepVMUnProtect with respect to helping semantic-based malware detection.To this end, we leverage an existing malware classification technique [11] to train a model using 5,606 benign and 5,148 malicious non-packed apps. The reasons to choose this technique are three-fold: *(a)* it performs malware detection based upon code semantics rather than syntactic features; *(b)* it models program semantics using opcode sequences, and uses a Convolutional Neural Network to capture intrinsic patterns in these sequences; and *(c)* the tool is open-sourced [36].

Note that an alternative option for building this detection model is to use packed malware and benign apps (i.e., vBytecode sequences) to train a malware classifier and then directly test encrypted unknown apps without unpacking. However, this is less practical as it requires to use a target VM-based packing service to generate many encrypted apps, and especially, malware instances.

We apply the trained classifier to the non-packed original version of 340 apps (283 malicious + 57 benign), and use the 297 true positives – 254/283 malicious and 43/57 benign apps – as the ground truth. Note that although this detector may misclassify 10% of the non-packed apps, this is orthogonal to our study. Our goal is to answer this question: *if an original app is correctly identified to be malicious or benign (true*

TABLE III: Semantic and Context Coverage of ML Models. ● for full coverage; ◐ partial coverage; ○ no coverage.

	JNI Semantics	Execution Context
Bi-LSTM + SA	○	●
LSTM + RNN + SA	●	◐
Bi-LSTM + RNN	●	◐
Bi-LSTM + RNN + SA	●	●

positive), after it has been packed using VMP, can its packed & unpacked version still be correctly detected?

As a baseline, we first directly send the 297 packed apps to the classifier. The results show that VM-packed malware can effectively evade semantic-based detection which simply considers any packed apps to be benign – as it is unable to find known malicious opcode sequences. Then, we use both our tool and **Parema** to unpack these 297 encrypted programs, and pass the recovered code to the classifier. Table II illustrates our results, which indicates that DeepVMUnProtect can effectively recover Dalvik opcode sequences to facilitate semantics-aware malware detection. A large majority (92.1%) of malware recovered by DeepVMUnProtect can be correctly flagged by the detector. In contrast, only 21.7% of the malware unpacked by **Parema** can trigger warnings. Meanwhile, **Parema** causes 3x false positives compared to DeepVMUnProtect when testing benign applications.

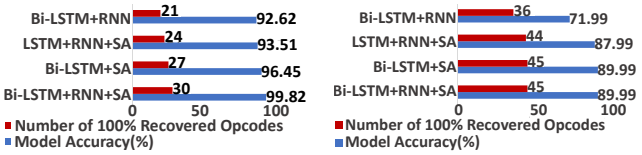
It turns out that false alarms can be caused by even slight misprediction of Dalvik opcodes. For instance, although we may merely misclassify a single `check-cast` (`0x1F`) as a `new-instance` (`0x22`), the nature of the corresponding opcode sequence suddenly changes – while the original sequence **`''1F 70 6E 14 6E 0C''`** is more likely to appear in benign samples, the occurrence of the predicted sequence **`''22 70 6E 14 6E 0C''`** in malware instances is 5 times higher than its occurrence in benign apps. As a result, this one-byte difference can lead to completely different detection results. Notice that these two opcodes in fact belong to the same opcode category defined by **Parema**, and thus cannot be differentiated by the prior work. This again shows that manually defined similarity metrics cannot be used to effectively enable malware detection.

D. Semantic and Context Awareness

Further, we expect to understand the effectiveness of our model, particularly how well JNI semantics and execution contexts aid opcode recovery. Thus, we perform an ablation study measuring individual model recovery accuracy. Specifically, we compare the four different models listed in Table III that partially or completely cover semantics and contexts. Particularly, Bi-LSTM plus Self-Attention captures full contexts, and RNN encodes JNI semantics.

To demonstrate the impact of JNI calls, we divide opcodes into two classes depending on whether their handlers are implemented using JNIs, and apply these models to both classes. Specifically, we randomly select 34 opcodes w/ JNIs and 48 opcodes w/o JNIs, and eventually test 86,180 instructions.

Figure 16 shows the comparison results. In general, for opcodes implemented either with or without JNIs, our fully semantic and context-aware model **Bi-LSTM + RNN + SA** always achieves the best performance. We have also observed that Self-Attention makes a major contribution to an accurate



(a) **Opcodes w/ JNIs** (b) **Opcodes w/o JNIs**

Fig. 16: Opcode-Level Accuracy for Different Models. Blue bars represent recovery accuracy; red bars indicate the amounts of opcodes that can be 100% recovered.

opcode recovery, as it selectively augments the impact of useful semantics and contexts. Nevertheless, Self-Attention alone cannot yield the highest accuracy – the JNI semantics and execution contexts matter. In particular, if an opcode is translated using JNI calls (Figure 16a), RNN is thus necessary to capture this strong signature. Hence, Bi-LSTM + **RNN** + SA outperforms Bi-LSTM + SA. Regardless of whether JNIs are used, **Bi**-LSTM + RNN + SA always outperforms LSTM + RNN + SA, as the former acquires the contextual knowledge which can only be obtained from backward analysis.

A further case study shows that our **Bi-LSTM + RNN + SA** model has indeed learned to capture the semantics and contexts of Dalvik opcodes: when our model observes the presence of this list of JNI calls – `NewStringUTF`, `CallObjectMethodV`, `ExceptionCheck`, `DeleteLocalRef`, `NewGlobalRef` – it will consider `const-string` to be the most relevant opcode with 100% confidence, while believing that other opcodes are very unlikely to be relevant.

E. Runtime Performance

Model training is a one-time effort and takes around 103 hours. We also spend 23 minutes to train the malware classifier. In contrast, packed app opcode sequence recovering and malware detection are faster. On average, it takes DeepVMUnProtect 5,616 seconds to collect all of the ARM traces while executing a VM-packed app, 51.45 seconds to recover the opcode sequence from the collected traces, and 1 second to classify whether the given app is malicious based on the recovered opcode sequence.

To quantify the failure rate of HWBPs, we analyzed the logs of 542 packed applications and recorded how often HWBPs failed to intercept the target code regions. On average, each application requires 33 runs for the HWBPs to be correctly positioned and each failed run takes only 0.48 seconds on average.

VII. Discussions

A. Large-Scale In-the-Wild Malware Study

Study & Findings To investigate VM-packed malware in the Android ecosystem, we conducted a large-scale study analyzing 16,417 malware samples, yielding several *noteworthy findings*. Our dataset was sourced from UNB [37], [38], VirusTotal [39], and Bazaar [40]. Details of this dataset are shown in Table IV.

Our analysis identified two VM-packed malware samples within the Bazaar dataset. Verification with DeepVMUnprotect confirmed their use of Qihoo and NMMP [41] packers, as outlined in Table V.

	UNB[34]	VirusTotal[39]	Bazaar[40]
Dataset Number	12,705	3,712	511
Updated Year	2017 & 2020	2018-2021	2024
Packed Number	1,007	538	18(**2 VM-pack**)

TABLE IV: Malware Datasets Used in the Study

	8a8fadfbd[42]	f9417e0345[43]
Packer	Qihoo	NMMP
Update Time	2024-10-05	2024-09-05
Original Country	US	IT
VirusTotal Result	7/66[44]	7/68[45]

TABLE V: Information on VM-Packed Malware Samples

As shown in Table V, these samples were uploaded by security analysts in the United States and Italy, suggesting identification through expert analysis rather than automated detection. Additional VirusTotal results indicated relatively low malware detection rates, with only 7 out of 66 and 7 out of 68 antivirus engines identifying them as malicious. This low detection score highlights a gap in current automated detection capabilities for VM-packed malware.

Conclusions The following conclusions can be drawn from our findings:

1) VM-packed malware exists in the wild, posing potential threats to mobile security.
2) Current malware detection engines struggle with VM-packed malware, as evidenced by the low detection rates on VirusTotal.
3) Effective detection of VM-packed malware remains highly reliant on manual analysis due to the limitations in automated detection methods.

These findings underscore the challenges in detecting packed malware, particularly VM-packed samples, within the Android malware detection domain. DeepVMUnprotect offers valuable support in bridging gaps within existing detection methods, especially for identifying VM-packed samples that evade conventional detection engines.

B. Limitations

Dynamic Coverage In DeepVMUnProtect, we recover the packed applications from their dynamic execution traces. As is well known, code coverage remains a challenging issue in dynamic analysis. To maximize coverage, we have attempted to switch between as many activities as possible using Frida. However, this approach may fail in certain cases, as constructing the correct intents to target specific activities is challenging in automated Android testing. Enhancing code coverage remains a potential area for future work in DeepVMUnProtect.

Selective Instruction Recovery In DeepVMUnProtect, some training data from the packer service is required to train the opcode semantics recovery model. This design choice is common in the analysis of VM-packed applications, both in the Android ecosystem [9] and in the x86-64 environment [46]. However, when the packing service is unavailable, recovering semantics from the packed application remains an unresolved issue, which could also be a future direction for DeepVMUnProtect.

Complete Bytecode Additionally, DeepVMUnProtect does not currently recover the operands of the packed application, as the opcode information alone suffices for malware detection. However, recovering complete bytecode (both opcodes and operands) could be beneficial in other contexts. We consider this as a potential area for future work.

Pure Native Packer In the current design, DeepVMUnProtect hooks `JNIMethodStart()` and monitors access to the encrypted code region, which is a suitable approach for analyzing current VM packers. However, attackers could potentially develop a pure native packer to evade DeepVMUnProtect. Specifically, instead of interpreting the smali instructions within a Java function, attackers could transform the Java function into a pure native function with the same semantics. In such a scenario, DeepVMUnProtect—along with all existing Android VM-based packed application analysis tools—would fail to analyze the application effectively. This is because the semantics of the original smali instructions would be entirely translated into native code, making it extremely challenging to divide the native code into clear semantic blocks and recover the original smali instructions. Therefore, the implementation of pure native functions could represent the next evolution of Android packers.

VIII. Related Work

Semantics-Based Android Malware Detection. Many efforts [12], [47], [8], [48], [49], [50], [51], [52], [11] have been made to automatically detect Android malware based upon code semantics. To extract program semantics, prior work has depended on opcode occurrence [48], opcode sequence [12], [47], [11], [49], data dependencies [8]. Semantics-based detection outperforms code pattern classification fundamentally for its robustness [8].

Binary Analysis via Deep Learning. Many prior studies have applied deep learning techniques to binary program analysis. To name a few, Shin *et al.* [20] first proved deep learning neural network (i.e., RNN) can be used to solve security problems such as function identification. Chua *et al.* proposed EKLAVYA [19] to recover the types signature of functions from binary code. Guo *et al.* [21] designed a new neural network architecture based on LSTM to facilitate the value set analysis. To apply the same idea to our context of Android malware analysis, we must address the challenge of modeling the Android app semantics and contexts.

Unpackers for VMP. A lot of research has been proposed to analyze VM-protected programs in traditional desktop systems. Monirul *et al.* [10] attempted to recover the semantics of VM-Protected malwar by reconstructing an entire packer VM and then extracting the semantics of each VM handler. Rolf [53] proposed a manual approach to reverse engineer packer VM. Coogan *et al.* [54] studied how binary instructions affect the interactions between systems and applications and used this as a guidance to identify original code. To deobfuscate the VM-protected code in a more general way, Babak *et al.* [55] proposed to examine the intrinsic dependencies between inputs and outputs. In addtion, Syntia [56] leveraged program synthesis to capture the semantics of execution traces. Though effective for desktops, applying unpacking

techniques to Android is non-trivial due to its unique paradigm and translation layer. The state-of-the-art **Parema** [9] unpacker still falls short, using unreliable heuristics and models.

IX. Conclusion

In this paper, we propose DeepVMUnProtect a deep learning-based approach to automatically and accurately capture the semantics of VM-packed code for semantic-based Android malware classification. To this end, we formally define the *context-aware opcode semantics* and invent a custom neural network model to systematically reconstruct opcode semantics. Experiments have shown that DeepVMUnProtect significantly outperforms the state-of-the-art tool.

References

[1] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang, "Things you may not know about android (un) packers: A systematic study based on whole-system emulation." in Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS'18), Feb 2018.

[2] D. Web, "Android packed malware 54525," https://vms.drweb.com/virus/?i=21885397.

[3] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: Toward extracting hidden code from packed android applications," in Computer Security – ESORICS 2015, 2015.

[4] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, "Appspear: Bytecode decrypting and dex reassembling for packed android malware," in Research in Attacks, Intrusions, and Defenses, 2015.

[5] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017.

[6] L. Xue, H. Zhou, X. Luo, Y. Zhou, Y. Shi, G. Gu, F. Zhang, and M. H. Au, "Happer: Unpacking android apps via a hardware-assisted approach," in Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P'21), May 2021.

[7] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in 2005 IEEE Symposium on Security and Privacy (S&P'05). IEEE, 2005, pp. 32–46.

[8] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '14, 2014.

[9] L. Xue, Y. Yan, L. Yan, M. Jiang, X. Luo, D. Wu, and Y. Zhou, "Parema: An unpacking framework for demystifying vm-based android packers," in 2021 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21), 2021.

[10] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in 2009 30th IEEE Symposium on Security and Privacy, 2009.

[11] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. Joon Ahn, "Deep android malware detection," in Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, ser. CODASPY '17, 2017.

[12] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in Detection of Intrusions and Malware, and Vulnerability Assessment, U. Flegel, E. Markatos, and W. Robertson, Eds., 2013.

[13] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "Droidsieve: Fast and accurate classification of obfuscated android malware," in Proceedings of the seventh ACM on conference on data and application security and privacy, 2017, pp. 309–320.

[14] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. J. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," in 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017. The Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/mamadroid-detecting-android-malware-building-markov-chains\-behavioral-models/

[15] H. Cai, "Assessing and improving malware detection sustainability through app evolution studies," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 29, no. 2, pp. 1–28, 2020.

[16] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 26, no. 3, pp. 1–29, 2018.

[17] V. M. Afonso, P. L. de Geus, A. Bianchi, Y. Fratantonio, C. Krügel, G. Vigna, A. Doupé, and M. Polino, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in Network and Distributed System Security Symposium, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:6506183

[18] "Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers," 2022, https://frida.re/. [Online]. Available: https://frida.re/

[19] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in 26th USENIX Security Symposium (USENIX Security 17). Vancouver, BC: USENIX Association, Aug. 2017, pp. 99–116. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chua

[20] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in 24th USENIX Security Symposium (USENIX Security 15). Washington, D.C.: USENIX Association, Aug. 2015, pp. 611–626. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/shin

[21] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, "DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis," in 28th USENIX Security Symposium (USENIX Security 19). Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1787–1804. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/guo

[22] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[23] "The lldb debugger," 2022, https://lldb.llvm.org/. [Online]. Available: https://lldb.llvm.org/

[24] "From research to production," 2022, https://pytorch.org/. [Online]. Available: https://pytorch.org/

[25] Baidu, "Baidu mobile platfrom," Website, 2022, https://app.baidu.com/newapp/index/.

[26] ijiami, "ijiami," Website, 2022, https://www.ijiami.cn/.

[27] Bangcle, "Bangcle," Website, 2022, https://www.bangcle.com/.

[28] maoabc, "Nmmp packer," 2023. [Online]. Available: https://github.com/maoabc/nmmp

[29] Anzhi, "Anzhi," Website, 2022, http://www.anzhi.com/.

[30] Baidu, "Baidu app store," Website, 2022, https://shouji.baidu.com/.

[31] Xiaomi, "Xiaomi app store," Website, 2022, https://m.app.mi.com/.

[32] Qihoo-360, "360 app store," Website, 2022, https://m.app.so.com/.

[33] "Android packer identifier," 2023, https://github.com/MagiCiAn1/APKProtectionSearch. [Online]. Available: https://github.com/MagiCiAn1/APKProtectionSearch

[34] A. H. Lashkari, A. F. A. Kadir, L. Taheri, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark android malware datasets and classification," in 2018 International Carnahan Conference on Security Technology (ICCST), 2018.

[35] Y. Zhou and X. Jiang, "Android malware genome project," Website, 2015, http://www.malgenomeproject.org/.

[36] "Deep android malware detection," 2022, https://github.com/niallmcl/Deep-Android-Malware-Detection. [Online]. Available: https://github.com/niallmcl/Deep-Android-Malware-Detection

[37] S. Mahdavifar, A. F. A. Kadir, R. Fatemi, D. Alhadidi, and A. A. Ghorbani, "Dynamic android malware category classification using semi-supervised deep learning," in IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress, DASC/PiCom/CBDCom/CyberSciTech 2020, Calgary, AB, Canada, August 17-22, 2020. IEEE, 2020, pp. 515–522. [Online]. Available: https://doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00094

[38] S. Mahdavifar, D. Alhadidi, and A. A. Ghorbani, "Effective and efficient hybrid android malware classification using pseudo-label stacked auto-encoder," J. Netw. Syst. Manag., vol. 30, no. 1, p. 22, 2022. [Online]. Available: https://doi.org/10.1007/s10922-021-09634-4

[39] "Virustotal," 2024, https://www.virustotal.com/. [Online]. Available: https://www.virustotal.com/

[40] "Malware bazaar detector," 2024, https://bazaar.abuse.ch/. [Online]. Available: https://bazaar.abuse.ch/

[41] "Github repo of nmmp," 2024, https://github.com/maoabc/nmmp. [Online]. Available: https://github.com/maoabc/nmmp
[42] "Qihoo vm packed malware," 2024, https://bazaar.abuse.ch/sample/8a8fadfbdeabcede6ac1a1798d47473652046c6309f8b922f16265d279c21b99/. [Online]. Available: https://bazaar.abuse.ch/sample/8a8fadfbdeabcede6ac1a1798d47473652046c6309f8b922f16265d279c21b99/
[43] "Nmmp vm packed malware," 2024, https://bazaar.abuse.ch/sample/f9417e0345634e89a4f808dc7f56eec9690c5f0445fd8672aa2a29ec01dff35b/. [Online]. Available: https://bazaar.abuse.ch/sample/f9417e0345634e89a4f808dc7f56eec9690c5f0445fd8672aa2a29ec01dff35b/
[44] "Virustotal report on qihoo vm packed malware," 2024, https://www.virustotal.com/gui/file/8a8fadfbdeabcede6ac1a1798d47473652046c6309f8b922f16265d279c21b99. [Online]. Available: https://www.virustotal.com/gui/file/8a8fadfbdeabcede6ac1a1798d47473652046c6309f8b922f16265d279c21b99
[45] "Virustotal report on nmmp vm packed malware," 2024, https://www.virustotal.com/gui/file/f9417e0345634e89a4f808dc7f56eec9690c5f0445fd8672aa2a29ec01dff35b. [Online]. Available: https://www.virustotal.com/gui/file/f9417e0345634e89a4f808dc7f56eec9690c5f0445fd8672aa2a29ec01dff35b
[46] S. Li, C. Jia, P. Qiu, Q. Chen, J. Ming, and D. Gao, "Chosen-instruction attack against commercial code virtualization obfuscators," 2022. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/auto-draft-210/
[47] P. O'Kane, S. Sezer, K. McLaughlin, and E. G. Im, "Svm training phase reduction using dataset feature filtering for malware detection," IEEE Transactions on Information Forensics and Security, vol. 8, no. 3, pp. 500–509, 2013.
[48] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features," IEEE Transactions on Information Forensics and Security, vol. 14, no. 3, pp. 773–788, 2019.
[49] S. Ni, Q. Qian, and R. Zhang, "Malware identification using visualization images and deep learning," Computers & Security, vol. 77, pp. 871–885, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404818303481
[50] B. Yuan, J. Wang, D. Liu, W. Guo, P. Wu, and X. Bao, "Byte-level malware classification based on markov images and deep learning," Computers & Security, vol. 92, p. 101740, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404820300262
[51] S. Alam, R. Horspool, I. Traore, and I. Sogukpinar, "A framework for metamorphic malware analysis and real-time detection," Computers & Security, vol. 48, pp. 212–233, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404814001576
[52] D. Vasan, M. Alazab, S. Venkatraman, J. Akram, and Z. Qin, "Mthael: Cross-architecture iot malware detection based on neural network advanced ensemble learning," IEEE Transactions on Computers, vol. 69, no. 11, pp. 1654–1667, 2020.
[53] R. Rolles, "Unpacking virtualization obfuscators," ser. WOOT'09. USA: USENIX Association, 2009.
[54] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: A semantics-based approach." New York, NY, USA: Association for Computing Machinery, 2011, p. 275–284.
[55] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in 2015 IEEE Symposium on Security and Privacy, 2015, pp. 674–691.
[56] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, "Syntia: Synthesizing the semantics of obfuscated code," Vancouver, BC, 2017, pp. 643–659.