Defining and Detecting Environment Discrimination in Android Apps

Yunfeng Hong[1], Yongjian Hu[2], Chun-Ming Lai[1], S. Felix Wu[1], Iulian Neamtiu[3], Patrick McDaniel[4], Paul Yu[5], Hasan Cam[5], and Gail-Joon Ahn[6]

[1] University of California, Davis
yfhong, cmlai, sfwu@ucdavis.edu
[2] University of California, Riverside
yhu009@cs.ucr.edu
[3] New Jersey Institute of Technology
iulian.neamtiu@njit.edu
[4] Pennsylvania State University
mcdaniel@cse.psu.edu
[5] U.S. Army Research Laboratory
{paul.l.yu, hasan.cam}.civ@mail.mil
[6] Arizona State University
Gail-Joon.Ahn@asu.edu

Abstract. Environment discrimination — a program behaving differently on different platforms — is used in many contexts. For example, malware can use environment discrimination to thwart detection attempts: as malware detectors employ automated dynamic analysis while running the potentially malicious program in a virtualized environment, the malware author can make the program virtual environment-aware so the malware turns off the nefarious behavior when it is running in a virtualized environment. Therefore, an approach for detecting environment discrimination can help security researchers and practitioners better understand the behavior of, and consequently counter, malware. In this paper we formally define environment discrimination, and propose an approach based on abstract traces and symbolic execution to detect discrimination in Android apps. Furthermore, our approach discovers what API calls expose the environment information to malware, which is a valuable reference for virtualization developers to improve their products. We also apply our approach to the real malware and third-party-researcher designed benchmark apps. The result shows that the algorithm and framework we proposed achieves 97% accuracy.

Keywords: Android, Malware Detection, Environment Discrimination

1 Introduction

In the past decade, the smartphone has replaced the PC as the most frequently-used Internet access device [1]. Along with the rising popularity of mobile devices, malware is also rapidly growing in terms of both quantity and sophistication.

For example, reports show that in the first quarter of 2017, 8,400 new malware samples were discovered every day [2], which results in the high demanding of malware detection and analysis. Dynamic analysis is a popular approach for analyzing application behaviors, and is usually deployed on virtual environments for performance and security reasons. However, malware authors are only interested in "real" phones used by actual customers. In contrast to the desktop/server platform, smartphone sandboxes have very limited use on mobile platforms (for both application development and dynamic analysis) because sensors which drive app behavior (such as GPS, camera, microphone) have to be mocked, which complicates development and analysis [3]. Thus, malware authors intentionally develop malware that detects the running environment and adjust malware behavior accordingly, as shown in Figure 1, when running on virtual environments, "smart" malware hides its suspicious behavior to evade dynamic analysis, and such behavior will be exposed when running on a real device. Some imparities between real devices and virtual machines such as CPU performance and battery consumption are difficult to be eliminated. Furthermore, it is infeasible to enumerate all heuristics that differentiate real devices and virtual machines. Thus, simple mitigation approach such as blacklist filtering is not capable to solve the problem, and a more fundamental and comprehensive approach is required to mitigate environment discrimination in Android applications.

Environment discrimination has been used in many other fields, besides dynamic analysis evasion. For example, some smartphone manufacturers detect when certain benchmarks are running and drive the CPU to maximum power in order to reach an edge in their benchmark ratings [4–6]. In another example from the automotive world, in certain Volkswagen models, the diesel engine controller software detects whether the car is running on a test bench, and changes engine parameters accordingly to subvert emission tests [7].

This paper has three major contributions: First, we formally define environment discrimination by leveraging the concepts of abstract specification and trace assertion [8–10]. Secondly, our work use these abstractions to construct an algorithm that is able to detect both already-known and unknown discriminating behaviors in linear time. Finally, by combining trace assertion with symbolic execution, our algorithm efficiently discovers the set of API calls that trigger environment discrimination: instead of exploring a potentially infinite set of execution paths as a static approach would do, our technique bounds the exploration space to permit efficient analysis: $O(n)$, where n is the size of the trace.

The evaluation result shows that the detection accuracy is 100% when the discrimination is executed during testing, and 97% for all test cases. We also show that the environment discrimination technique is not widely employed in the real malware, but as emulation becomes more and more common over time, we will see more discrimination behaviors in the future.

Note that a program discriminating the environment does not necessarily imply malicious intent. Benign programs can behave differently in different environment as well. For example, Google Maps behaves differently in a virtual environment compared to a real device due to lack of GPS in a virtual setting.

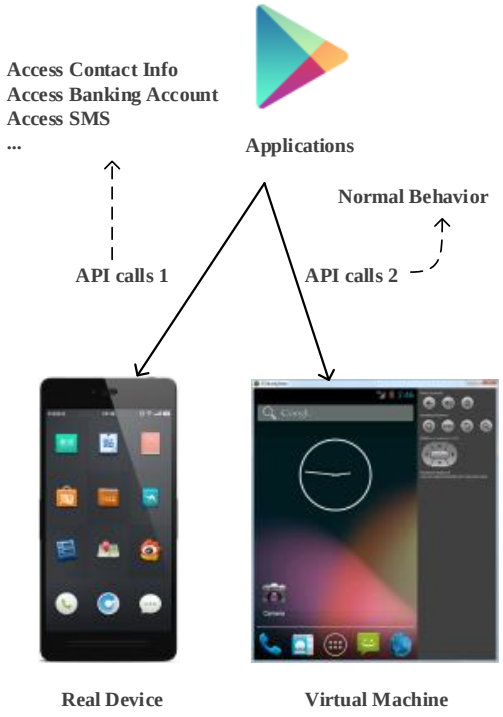

Fig. 1: Example of an application discriminating between virtual machine and real device to evade dynamic analysis.

Accordingly, this paper focuses on detecting environment discrimination, e.g., as employed by malware, but we do not attempt to detect malicious behavior per se.

The rest of this paper is organized as follow: Section 2 gives the definition of environment discrimination and the algorithm in theory. The application of definition and algorithm is illustrated in Section 3. Section 4 discusses the time complexity and robustness of our algorithm. Section 5 and Section 6 present related work and conclude the paper, respectively.

2 Definition of Environment Discrimination

In this section, we first explain the concept of trace equivalence and trace abstraction, and then discuss the relevance of the two concepts in defining environment discrimination. Finally, we describe symbolic execution against a trace for finding discriminating contributors.

This section proposes the theoretical background of the environment discrimination detection of Android app in Section 3. However, readers who do not want to dive into theory may directly jump to Section 3 without concern.

2.1 Trace Equivalence

A trace of a program, which is a description of a sequence of calls on functions starting with the program in initial state, consists of *O-functions* and *V-functions* [8]. *V-functions* return values that give information about parts of program, while *O-functions* only change internal data. To begin, we formalize function calls F and traces T.

A function call F consists of its name, parameter list, and return values. Return values are always empty in *O-functions*. Two calls, F_1 and F_2, are equivalent if and only if all three parts are exactly the same (denoted as $F_1 \equiv F_2$; we will describe this check in detail in Section 2.2). A trace T is described by the following syntax:

$$
\begin{aligned}
&< T > \quad ::= \{< subtrace >\}. < tailtrace > \\
&< subtrace > ::= \{< O - function >\}. < V - function > \\
&< tailtrace > ::= \{< O - function >\}
\end{aligned}
$$

$\{*\}$ *represents any number of occurrences of* $*$.

F_{ijk} is the k^{th} function call in j^{th} subtrace in i^{th} trace. The definition of the size of trace T is the number of subtraces and tailtrace, denoted as $|T|$. S is a *subtrace.*

A trace T is *legal*, denoted $\lambda(T)$, if the functions in T will not result in a trap. Note that an empty trace is always legal ($\lambda(\_) = true$); and the prefix of any legal trace is always legal, i.e., $\lambda(T.S) = true => \lambda(T) = true$.

If $\lambda(T.X) = true$, and X is a syntactically correct *V-function* call, $V(T.X)$ describes the value returned by X after the execution of T.

Trace specification consists of syntax and semantics. The syntax provides the name, parameter types and return value types of each function. The semantics comprises of three types of assertions: (1) *legality assertions* which describe how to call functions that will not result in a trap; (2) *equivalence assertions* which specify a set of equivalence relations in traces; and (3) *V-function assertions* expressed in terms of values returned by *V-functions.*

We now exemplify these trace concepts by providing Bartussek and Parnas [8] integer stack specification.

Syntax

$PUSH :< integer > \times < stack > \rightarrow < stack >$
$POP :< stack > \rightarrow < stack >$
$TOP :< stack > \rightarrow < integer >$
$DEPTH :< stack > \rightarrow < integer >$

Legality

$\lambda(T) => \lambda(T.PUSH(a))$
$\lambda(T.TOP) <=> \lambda(T.POP)$

Equivalences

$T.DEPTH \equiv T$
$T.PUSH(a).POP \equiv T$

$\lambda(T.TOP) => T.TOP \equiv T$

Values

$\lambda(T) => V(T.PUSH(a).TOP) = a$

$\lambda(T) => V(T.PUSH(a).DEPTH) = 1 + V(T.DEPTH)$

$V(DEPTH) = 0$

The "equivalence" in the above specification is a set of assertions defining the semantics of the trace specification, while "trace equivalence" in environment discrimination indicates that the behavior of a program in two environments is not distinguishable from two traces.

Definition 1 *TRACE EQUIVALENCE*

Given 2 traces T_1 and T_2, we claim T_1 is *equivalent* to T_2 (denoted as $T_1 \equiv T_2$) when all the following conditions hold:

(1) Both T_1 and T_2 contain tailtraces or neither one contains a tailtrace.
(2) T_1 and T_2 have the same number of subtraces n.
(3) For each pair of subtraces in T_1 and T_2, we formalize subtrace T_{ij} ($i = 1$ *or* 2, $1 \leq j \leq n$) as:

$$T_{ij} ::= O_{ij1}...O_{ijk}...O_{ijo-1}V_{ij}$$

where o is the length of T_{ij}, and $0 \leq k < o$.
For each pair of subtraces T_{1j} and T_{2j}, where p, q are the lengths of T_{1j} and T_{2j}:
(i) $\lambda(T_{1jp-1}.V_{2j}) = true$
(ii) $\lambda(T_{2jq-1}.V_{1j}) = true$
(iii) $V(T_{1j}) = V(T_{2j})$

Ideally, the legality is ruled by a set of assertions so that all λ expressions above are checked through the pre-defined assertions. However, in practice, it is infeasible to provide a complete set of legality assertions. Thus, we either ignore the legality rules above or enforce the name of *V-function* of subtraces are identical if the legality assertions are not available. Both approaches admit that $\lambda(T_{1jp-1}.V_{2j})$ and $\lambda(T_{2jq-1}.V_{1j})$ are true by default.

We show an example to further illustrate trace equivalence: A stack for integer values S_1 is specified as previously discussed. The other stack S_2 is similar to S_1. The only difference between S_1 and S_2 is the size of each element. Specifically, the size of each element in S_2 is 2 instead of 1:

$\lambda(T) => V(T.PUSH(a).DEPTH) = 2 + V(T.DEPTH)$

Assume that a program P, defined next, runs on S_1 and S_2, respectively:

```
1   PUSH (1)
2   PUSH (2)
3   if (TOP == 1)
4     PUSH (3)
5   POP
6   if (DEPTH == 2)
7     PUSH (4)
```

```
8     PUSH (5)
9     TOP
10    DEPTH
11  else
12    PUSH (6)
13    TOP
14    TOP
15  POP
```

We denote the traces generated by the two executions as T_1 and T_2:
$T_1 ::= PUSH(1).PUSH(2).TOP.POP.DEPTH.PUSH(6).TOP.TOP.POP$
$T_2 ::= PUSH(1).PUSH(2).TOP.POP.DEPTH.PUSH(4).PUSH(5)$
$.TOP.DEPTH.POP$

T_1 contains 4 subtraces: $PUSH(1).PUSH(2).TOP$, $POP.DEPTH$, $PUSH(6).TOP$, and TOP along with a tail trace: POP. T_2 also contains 4 subtraces: $PUSH(1).PUSH(2).TOP$, $POP.DEPTH$, $PUSH(4).PUSH(5).TOP$, and $DEPTH$ along with a tail trace: POP. Based on these results, conditions (1) and (2) in Definition 1 hold. T_{11} and T_{21} have the same *V-functions* and return values: 2. However, $V(T_{12}) = 1$ and $V(T_{22}) = 2$ which violates condition (3)(iii). The pairs T_{13} & T_{23} and T_{14} & T_{24} violate condition (3) as well; therefore $T_1 \not\equiv T_2$.

The definition of trace equivalence reveals the equivalent relation of 2 executions from the observation of traces. Note that two equivalent traces are not necessarily identical. For example if we define
$T_3 ::= PUSH(1).POP.DEPTH$
$T_4 ::= DEPTH$
then $T_3 \equiv T_4$ but they are not identical.

2.2 Trace Abstraction and Defining Environment Discrimination

Given two specific traces, Definition 1 is an effective tool for determining execution equality, or semantic similarity. However, finding a proper trace is a challenge. Thus, we propose *trace abstraction* as a procedure for checking $T_1 \equiv T_2$ efficiently. Algorithm 1 (described shortly) lists the steps of trace abstraction. Before introducing the algorithm, the concept of LCCS is introduced.

We first define the *longest common call subsequence* (LCCS), which is similar to the longest common substring (LCS) but replaces characters with function calls. LCCS is defined within the boundary of a subtrace or tail trace (but not the whole trace). To observe discriminating behaviors, we are interested in how a program reacts after a particular return value is obtained.

Parameter lists in function calls will be ignored. Consider how parameters can potentially influence the execution path of a program: given a pair of traces, if a function call returns the same return value regardless of different parameters, the parameter has no effect on the execution path by calling functions. On the other hand, if different parameters cause different return values, we are still able

to observe differences by examining return values. Another question worth taking into account is where the different parameters come from. One possible answer is that they are derived from previous different return values. It is interesting to find that even a randomized program can be reduced to this answer because the program always has to call a function to derive the random value. Another possible answer is that the source of different parameters is not captured in the trace, which is not discussed in this paper. Thus, the partial order in a function call abstraction is shown below:

$$ignore\ return\ value \leq ignore\ function\ call$$

The aforementioned partial order indicates that ignoring the function call is more abstract than ignoring the return value.

Algorithm 1 describes the procedure of trace abstraction: the algorithm takes two traces T_1 and T_2 as input and returns a new trace T, which is the abstraction of the input traces. We use the T_1 and T_2 from the previous subsection to illustrate the algorithm.

Algorithm 1 abstraction for two traces

```
 1: function MAIN(T1,T2)
 2:     Tsubtrace1 :=GET SUBTRACE LIST(T1)
 3:     Tsubtrace2 :=GET SUBTRACE LIST(T2)
 4:     T:=empty trace
 5:     Tsubtrace :=LCSS(Tsubtrace1, Tsubtrace2)
 6:     i:=0                                   ▷ i is the index of Tsubtrace
 7:     for each pair of subtraces S1, S2 in Tsubtrace do
 8:         T := T + ABSTRACT(S1, S2)
 9:     end for
10:     Ttail:=LCCS(T1tail,T2tail)
11:     T := T + Ttail
12:     return T
13: end function
14:
15: function ABSTRACT(subtrace1, subtrace2)
16:     T0:=LCCS(subtrace1, subtrace2)
17:     subtrace1:=subtrace1-T0
18:     subtrace2:=subtrace2-T0
19:     T1:=LCCS(subtrace1, subtrace2)
20:     T:=IN ORDER MERGE(T0, T1)
21:     return T
22: end function
```

The Algorithm starts with the MAIN() function; lines 2 and 3 cut T_1 and T_2 into subtraces, splitted by *V-function*s. Line 5 finds the Longest Common Subtrace Subsequence (LCSS) by only matching the function name of *V-functions*. In this example, $T_{subtrace}$ will be $\{T_{11}, T_{21}\}$, $\{T_{12}, T_{22}\}$, and $\{T_{13}, T_{23}\}$. Subtraces

that are not in $T_{subtrace}$ will not appear in T. The **for** loop on lines 7–9 calls ABSTRACT() for each pair of subtraces in $T_{subtrace}$ and reassembles them into T. Lines 10 and 11 find the LCCS for the tail trace and append T_{tail} to the end of T.

ABSTRACT() finds the minimal abstraction making two subtraces equivalent. T_0 is the LCCS for two traces without ignoring return values or function calls. In our example, when $subtrace_1$ and $subtrace_2$ are T_{13} and T_{23}, $T_0 = PUSH$. Lines 17 and 18 remove the function calls which appeared in T_0. Thus, $subtrace_1 = TOP$ and $subtrace_2 = PUSH.TOP$ after line 17 and 18 are executed. T_1 is the LCCS ignoring return values, $T_1 = TOP$. Line 20 merges T_0 and T_1 in order, ABSTRACT() returns subtrace: $PUSH.TOP$. Similarly ABSTRACT() returns $PUSH.PUSH.TOP$ for $\{T_{11}, T_{21}\}$ and $POP.DEPTH$ for $\{T_{12}, T_{22}\}$. Finally, $T := PUSH.PUSH.TOP.POP.DEPTH.PUSH.TOP.POP$.

Definition 2 *ENVIRONMENT DISCRIMINATION*

If $T_1 \equiv T_2$ under the abstraction of trace T, we say that program P does not discriminate the two environments under the abstraction of trace T. Program P does not discriminate both environments when $T_1 \equiv T_2$ without any abstraction ($T_1 \equiv T_2 \equiv T$).

It is clear that T is not guaranteed to hold its original trace specification; rather it is designed to capture as many common parts in the two executions as possible.

Finally, we claim that minimum abstraction T gives the lower bound of abstraction. Thus, the alias of T is T_{low}. Any abstraction that is finer grained (less abstract) than T_{low} cannot guarantee the equivalence relation between T_1 and T_2. The upper bound of abstraction assuring the correctness of detecting environment discrimination is not ignoring the function calls before p_0, denoted as T_{up}. Any abstraction T' holding $T_{up} \geq T' \geq T_{low}$ is acceptable to detect environment discrimination and its contributor.

2.3 Finding Discrimination Contributors with Symbolic Execution

In order to find relevant/discriminating function calls (i.e., calls that expose environment information leading to programs behaving differently) accurately and efficiently, we propose the use of symbolic execution against traces.

Symbolic execution [11, 12] is widely used in software engineering for generating test inputs, e.g., to explore different execution paths. One major limitation of symbolic execution is path explosion. Symbolically executing all program paths cannot scale to large programs because the number of paths grows exponentially with the number of conditional statements encountered: $\Theta(2^n)$, where n is the number of conditional statements encountered. When applying symbolic execution to find the subset of $\{S\}$, our algorithm matches the execution path with the trace to avoid path explosion.

We come back to the example of program P in Section 2.1 to illustrate symbolic execution against the trace. In order to find the discrimination contributor, the symbolic executor needs to run against T_1 and T_2, respectively. When the symbolic executor runs against T_1, initially, symbolic $\sigma = \emptyset$, path constraint $PC = true$, and a pointer ptr is pointing to the first function call in trace T_1. Whenever a function call is encountered, the algorithm checks the consistency between the current function call and the function ptr is pointing to in T_1. If they are consistent, ptr moves one function forward and symbolic execution continues. Thus, after line 2 is executed, $ptr \rightarrow TOP$. Every time a *V-function* is executed, the return values will be marked as a symbolic variable. TOP in line 3 leads to $\sigma = \{TOP_{line3} \rightarrow 2\}$. When if is executed, $PC \rightarrow TOP_{line3} = 1$, which is the constraint of basic block on line 4. $PC' \rightarrow \neg TOP_{line3} = 1$, $ptr \rightarrow POP$. During the execution of line 4, $PUSH$ and POP do not match. Thus, the current branch is not executed in the trace. As a result, PC' will be accepted as PC; ptr stays still, and the rest of the current branch does not need to be executed as well. When line 6 is executed, $PC \rightarrow DEPTH_{line6} = 2 \wedge \neg TOP_{line3} = 1$, which covers the basic block on lines 7–10. When the program symbolically executes to the diverge point of T_1 and T_2 (line 7 and line 12 in our example), $PC_{T_1} \rightarrow DEPTH_{line6} = 2 \wedge \neg TOP_{line3} = 1$. Similarly, $PC_{T_2} \rightarrow \neg DEPTH_{line6} = 2 \wedge \neg TOP_{line3} = 1$ after running symbolic executor against T_2. The discrimination contributors are the variables in the pairs of terms that are exactly reversed in 2 path constraints, which is $DEPTH$ in our example.

The time complexity of the algorithm to detect discriminating behavior and find contributors is $O(n)$, where n is the size of trace.

3 Detecting Environment Discrimination on Android

In this section, our theory is applied to detect environment discrimination on the Android platform. Before illustrating our approach, we introduce a prototype malware: Pi Calculator. To detect its inappropriate behavior, we first find an appropriate abstraction of the trace, and by applying the concept of trace equivalence, the algorithm is able to determine whether an application is behaving differently in two environments. Finally, with the help of symbolic execution, the algorithm finds discrimination contributor.

The procedure of detecting environment discrimination is shown in Figure 2. The trace collector first collects traces from the emulator ($T_{emulator}$) and the real device (T_{device}), then checks their equivalence under proper abstraction (discussed in detail in Section 3.2). If $T_{emulator} \not\equiv T_{device}$, the discrimination contributor is found by performing symbolic execution against $T_{emulator}$ and T_{device}.

3.1 Pi Calculator: An Environment-discriminating Malware

We have developed a prototype malware, Pi Calculator, that discriminates environments. It is a CPU benchmark application that evaluates CPU single-core

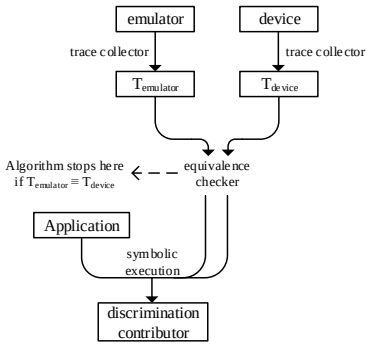

Fig. 2: Overview of our approach.

performance by recording the time it takes to calculate π to 5,000, 10,000, 15,000, or 20,000 digits.

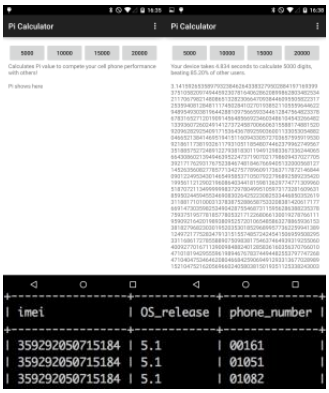

Fig. 3: Pi Calculator screenshot.

Device performance is reflected in the duration of the calculation (a slower device takes longer to compute π), which is uploaded to a remote database. We compute a *device score* by dividing the number of devices slower than current device by the number of all devices in the database. However, as a malware, Pi Calculator also uploads contact information to the server without informing the user. This application has bypassed the security check provided by one of the major Android markets. Due to privacy concerns, we only upload the first 5 digits of phone numbers (we do not upload the actual contact's name).

Table 1 presents the calculation time on various devices. Note that the CPU performance of the Android emulator is much higher than that of ARM-based real devices. As we mentioned before, dynamic analysis is usually performed on virtual machines. To evade dynamic analysis, Pi Calculator takes advantage of

Device	5,000 Digits		10,000 Digits		15,000 Digits		20,000 Digits	
	Avg (s)	StDev (s)	Avg (s)	StDev (s)	Avg (s)	StDev (s)	Avg (s)	StDev (s)
Emulator (Linux)	1.258	0.020	4.937	0.007	10.044	0.016	17.521	0.023
Mi Note Pro (5.0.2)	2.317	0.058	9.246	0.040	21.193	0.159	39.428	0.235
Google Nexus 6 (5.0.1)	4.304	0.017	18.335	0.022	41.352	0.179	74.398	0.538
Meizu MX3 (5.0.1)	4.970	0.168	19.916	0.480	48.055	1.112	85.559	0.841

Table 1: Pi Calculator calculation time. Note that real devices are much slower than the emulator.

this phenomenon and determines its running environment as a real device if it takes more than 2, 6, 15, 25 seconds to compute π to 5,000, 10,000, 15,000, and 20,000 digits, respectively. The app will access the contact list and upload all information to remote server only when a real device is detected but will not collect contact information when a virtual environment is detected.

The rest of this section explains how our algorithm detects the discrimination behavior employed by Pi Calculator in detail.

3.2 Detecting Environment Discrimination and Contributors

Selecting a proper trace is a key factor that determines the success and efficiency of discrimination detection. Our trace collector collects 3 kinds of function calls: application internal function calls, application calling API, and API internal function calls. Removing API internal calls is important because API internal calls may introduce non-determinism. Application internal call is also removed because it is not our interest. The two generated trace are denoted as T_{device} and $T_{emulator}$.

Instead of applying minimum abstraction T, we abstract T_{device} and $T_{emulator}$ by ignoring all method return values, denoted as T', and then check trace equivalence. Accuracy is guaranteed because $T_{up} \geq T' \geq T_{low}$. Almost all Android API calls have return values. So we regard all API calls in application level as *V-functions*. To apply the definition of environment discrimination, we need to check the definition of trace equivalence (Definition 1): Condition (1) of trace equivalence holds because both traces do not have a tail trace. Condition (2) is determined by whether two traces have the same size. The legality conditions ((i) and (ii)) in condition (3) are always true because there is no *O-function* in the trace. Thus, the trace equivalence check in our case simply reduces to comparing whether two traces call the same functions in order.

Now we consider composing symbolic execution with traces. As shown in Figure 4, despite different approaches, in order to discriminate, a program first collects information about its environment and determine the environment based on the collected data. Next, the program will behave differently according to the environment, which is the diverge point. We emphasize that information collection and computation are not necessarily in order, and can even be mixed.

Thus, we split the trace into two parts. The first part is information collection and computing, and the second part is the divergent part. In this section, we only focus on the first part, that is, $T_{\rightarrow p0}$. Specifically, the symbolic executor runs against $T_{device \rightarrow p0}$ and $T_{emulator \rightarrow p0}$.

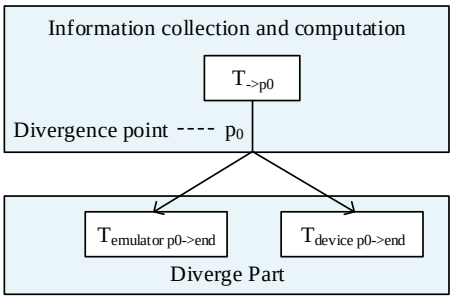

Fig. 4: Relationship between execution and trace.

The following rules describe the process of execution.

- Initially, *ptr* points to the first call in T_{device} / $T_{emulator}$.
- Whenever symbolic execution encounters an API call, the algorithm checks if the API is the same to the API that the pointer is pointing to in T_{device} / $T_{emulator}$. If not, it indicates this particular branch is not executed in trace, and we mark the PC belonging to that branch as false. If yes, we move the pointer to the next API call and continue executing.
- In T_{device} and $T_{emulator}$, the first pair of API calls after $T_{\rightarrow p0}$ actually are the first pair of API calls in two branches resulting from environment discrimination. Thus, symbolic execution runs until reaching p_0. ($T_{device \rightarrow p0}$ / $T_{emulator \rightarrow p0}$)
- The discrimination contributors are the variables in the pairs of terms that are exactly reversed in $PC_{T_{device \rightarrow p0}}$ and $PC_{T_{emulator \rightarrow p0}}$

We use Pi Calculator as an example to illustrate the procedure of finding discrimination contributor in detail.

In Figure 5, each box is an API method call. In each call, the first field is the method name, the second field is the method ID, and the third field is the class name the method belongs to; <init> indicates a constructor call. We are not able to locate a unique function call solely by method name because different classes might have methods that have the same method name, so method ID and method declaring class help us recognize a unique method call. During one execution, each method has a unique method ID. However, the same method usually owns a different ID in two executions. We determine a method call in different traces by matching both the method name and method declaring class. For instance, $< init >$ $(0x70c25f20, GetTime.java)$ is called twice in T_{device}, and this constructor is also called in $T_{emulator}$, which is $< init >$

Fig. 5: Two traces generated from Pi Calculator.

$(0x70ab5c20, GetTime.java)$, although the method ID is different. The algorithm determines that Pi Calculator discriminates the environment by finding a different pair of method calls: $< init > (0x75471a58, ContactsContract.java)$ and $< init > (0x708e92b0, DefaultHttpClient.java)$, which is where p_0 located.

Below is the code segment from Pi Calculator. During the course of symbolic execution against $T_{device \rightarrow p0}$, symbolic state σ and path constraint PC_{device} are maintained. All return values from API call will be marked as symbol. When line 4 and 5 are executed, $\sigma = \{start\_time \rightarrow start\_time_0, end\_time \rightarrow end\_time_0\}$. *start* has not been added to σ until line 7 because it receives a return value from $getCurrentNetworkTime$. At the end of line 12, $\sigma = \{start\_time \rightarrow start\_time_0, end\_time \rightarrow end\_time_0, start \rightarrow start_0, end \rightarrow end_0, result\_text \rightarrow null, time \rightarrow start_0 - end_0\}$, and $PC_{device} = \phi$. Note that symbolic execution does not execute a path that is not reflected in trace. In our example, lines 4 and 5 match the first two method calls in the trace. Similarly, lines 7 and 10 match the third and fifth calls in the trace. Line 9 matches the fourth call but we assign $result\_text$ as *null* because $setText$ does not have a return value. As mentioned before, all API calls are regarded as *V-functions*. Assigning *null* to $setText$ does not influence the result. $0x7074c2b0$ is recorded from line 45 that is called from line 13. In line 14, PC_{device} is updated to $PC_{device} = \{end_0 - start_0 > 2000\}$, corresponding to the basic block on line 15. The **if** branch line 19 is satisfied. Line 32, called by line 20, matches $< init > (0x75471a58, ContactsContract.java)$ in T_{device}. The symbolic execution stops because $0x75471a58$ is where p_0 located. $PC_{device} = \{end_0 - start_0 > 2000 \wedge real\_device = true\}$ Similarly, after running symbolic execution against $T_{emulator}$, $PC_{emulator} = \{\neg end_0 - start_0 >$

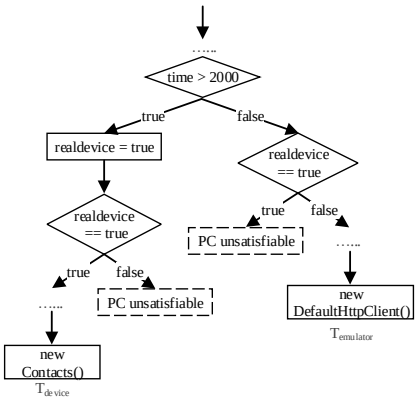

Fig. 6: Execution Tree of Pi Calculator.

$2000 \wedge \neg real\_device = true\}$. Following the rules in Section 2.3, discrimination contributors are 2 calls to: **getCurrentNetworkTime()**. Specifically, Pi Calculator determines its running environment by measuring the time it takes to calculate π. If the calculation takes less than 2 seconds, Pi Calculator regards its environment as an emulator. The execution tree is illustrated in Figure 6.

```
1   public class option extends ActionBarActivity {
2     private void clicked (int digit) {
3     ...
4       GetTime start_time = new GetTime();
5       GetTime end_time = new GetTime();
6       boolean realdevice = false;
7       start = start_time.getCurrentNetworkTime();
8       String pi = my_calculator.get_pi(digit);
9       result_text.setText(pi);
10      end = end_time.getCurrentNetworkTime();
11      time = end - start;
12      if (digit == 5000) {
13        new upload_score_5000().execute();
14        if (time > 2000) {
15          realdevice = true;
16        }
17      }
18      else {...}
19      if (realdevice == true) {
20        readAllContacts();
21      }
22      lower = new grab_lower_score().execute((long)digit, time).get();
23    }
24    private class upload_contacts extends AsyncTask<Contacts, Void, Void> {
25      protected Void doInBackground (...) {
26        HttpClient httpclient = new DefaultHttpClient();
27          ...
28      }
29    }
30    public void readAllContacts() {
31      while (cursor.moveToNext()) {
32        Contacts cur = new Contacts();
33        ...read each contacts info...
34        new upload_contacts().execute(cur);
35      }
36    }
```

```
37    private class grab_lower_score  extends AsyncTask<Long, Void, Integer> {
38      protected Void doInBackground (...) {
39        HttpClient httpclient = new DefaultHttpClient();
40        ....
41      }
42    }
43    private class upload_score_5000 extends AsyncTask<Score, Void, Void> {
44      protected Void doInBackground (...) {
45        HttpClient httpclient = new DefaultHttpClient();
46        ....
47      }
48    }
49  }
```

Note that the key advantage of combining symbolic execution with trace is that this combined analysis shrinks down the time complexity of symbolic execution since we do not execute branches that do not appear in the trace.

4 Evaluation and Discussion

4.1 Practical Malware Evaluation

We apply the framework described in Section 3 to 18 real world malware. They come from 10 different apps along with their variants. The result shows that none of the 18 apps discriminates the virtual machine and real device before exposing anomalous behavior, which indicates that the discrimination technique has not been widely applied by the malware developers. Even though discrimination behaviors cannot be found in a while, as emulation becomes more and more common over time, we will see more discrimination behaviors in the future.

Table 2: List of practical malware evaluated

App name	Number of variants
DroidKungFu4	3
FakeNetflix	1
Geinimi	1
GGTracker	1
GingerMaster	2
SndApps	2
Tapsnake	2
zHash	2
NickySpy	2
HippoSMS	2

Our detection algorithm is efficient. Checking environment discrimination behavior is in linear time, $O(n)$, where n is the size of trace. The time complexity of detecting discrimination contributors is also $O(n)$, where n is the number of lines of code. The reason is that running symbolic execution against trace matching shrinks down the time complexity of symbolic execution because executer will not execute a path if it cannot be found in trace.

4.2 Benchmark Malware Evaluation

Because environment discrimination is not widely applied in practical malware, we invite the third party researchers who have no knowledge in our algorithm injecting the environment discrimination code into the practical malware, and evaluate the framework against the test bench to perform a blind testing. In particular, the benchmark contains a set of malware injected with environment discrimination code with varieties of heuristics and malware that does not have such behaviors.

Table 3: Benchmark Malware Set Evaluation Result

Heuristics	# of Apps	Apps discriminate during execution	Detection Rate	Contributor Detection Rate	Accuracy
Property (API) heuristics	5	5	100%	100%	100%
File heuristics	5	5	100%	100%	100%
Component heuristics	5	5	100%	100%	100%
Sophisticated heuristics	5	4	80%	80%	80%
No discrimination	10	0	100%	N/A	100%
Overall	30	19	N/A	N/A	97%

Table 3 lists the evaluation result. 30 apps contained in the benchmark are categorized into 5 categories. The property (API) heuristics take advantage of the API call artifacts such as `getDeviceId()` and `Build.MODEL()`. The apps leveraging file and component heuristics check the existance of a specific file or hardware component, respectively. Sophisticated heuristics are more difficult to detect. For example, one app in benchmark tests whether the call log is empty. Another app checks whether the battery is always charging and remaining at 50%, which is the default configuration in most emulators. The last category is a set of apps without any discrimination behaviors for us to evaluate the false positive.

The overall accuracy is 97%, and the only case that fails is an app leveraging the time bomb to discriminate. The time bomb is not exposed during evaluation, thus it is not captured in the trace. Also, even though the false positive is 0%, it is not guaranteed that some discrimination behavior is not intended to differentiate the virtual machine and physical device. For example, false positive may occur when Google Map behaves differently as no GPS signal is found, and virtual machine usually does not provide location information if not configured. However, in this paper, we do not attempt to differentiate the intention of discrimination.

The major limitation of this work is that the detection framework will never be able to detect the discrimination behavior if such behavior is not captured in trace. For instance, our framework failed to detect the time bomb planted in one of the benchmark app because the time bomb was not triggered during the process of trace collection. Even though the time bomb does not directly dif-

ferentiate the virtual environment from the physical device, malware developers understand the time of malware being tested by security analysts is significantly shorter than the time of the app used by a real user. One potential approach to mitigate this problem is to run static analysis and generate all potential traces. As a trade off, this approach may bring false positive and the runtime can be up to $O(2^n)$, where n is the length of program.

5 Related Work

5.1 Dynamic and Tainting Analysis

Many dynamic analysis tools have been developed to analyze malware. This section cites and introduces the dynamic analysis works often used in either industry or academic. DroidScope[13] is a virtualization-based Android malware analysis platform, which reconstructs the OS-level and Java-level semantics seamlessly and simultaneously. Various analysis tools is also developed on top of DroidScope to collect native and Dalvik instruction traces, profiling API-level activity, and tainting analysis. TaintDroid [14] is an efficient and system-wide dynamic taint tracking and analysis system capable of tracking multiple sources of sensitive data, which leverages different levels of instrumentation to perform the analysis. Even though TaintDroid introduces only 14% overhead, modifying the components of Android exposes TaintDroid to some detection and evasion techniques [15–17]. Andrubis [18] combines static analysis with dynamic analysis on both Dalvik VM and system level, as well as several stimulation techniques to increase code coverage, which is built based on TaintDroid [13] and DroidBox [19].

Besides the tools introduced above, many other dynamic analysis tools have been developed to analyze malware, most of which extract API calls or system calls [20–24]. Several dynamic analysis tools record traces with in-guest technologies such as Norman Antivirus Sandbox [22] and CSSandbox [20]. Tools such as Ether [23] and HyperDBG [25] are implemented based on hardware-supported virtualization technology. However, for convenience and security reasons, more dynamic analysis tools are deployed on virtual environments. For instance, Google Boucer [26], VMScope [27] and TT-Analyze [21] are based on QEMU [28], which is a popular virtual machine.

5.2 Virtual Machine Evasion

On traditional platforms such as PCs, dynamic analysis systems are usually built based on virtualization. Consequently, PC malware developers design malware that is aware of virtual environments and exhibit benign behavior in such cases [29–31]. However, virtualization has matured in recent years. Many users even migrate physical environments to virtual instances, e.g., as in the Cloud, hence malware that discriminate virtual environments, stand to lose a large number of victim systems.

On the other hand, the application of virtualization on mobile platforms is quite limited. A normal user is very unlikely to run a mobile OS in a virtual

environment but dynamic analysis does, as we mentioned before. Recent work has shown that malwares on mobile platforms discriminate running environments to evade dynamic analysis based on virtualization [32, 3].

Few efforts have focused on environment discrimination. Morpheus generates heuristics to detect Android emulators and classifies heuristics as file, API, and system property [33]. BareCloud automatically detects evasive malware by using hierarchical similarity-based behavioral profile comparison; profiles are collected by running a malware sample in bare-metal, virtualized, emulated, and hypervisor-based analysis environments [34]. Balzarotti's paper is similar to our project [35], which also collects and compares the trace to find split personalities in malware. However, our work formally defines environment discrimination and employs symbolic execution against trace to find the discrimination contributors, which differentiates from Balzarotti's work.

6 Conclusion

The concept of environment discrimination has been applied in many areas. Dynamic analysis is a convenient and efficient approach to analyze program behavior, but some malware is able to detect the existence of virtual environments and evade detection. Some detection strategy such as evaluating hardware performance is infeasible to block in practice.

In this work, we define environment discrimination and an efficient algorithm to detect and describe such behavior. The time complexity to detect discrimination behavior and discrimination contributor is $O(n)$. The framework we proposed reaches 97% detection accuracy when testing against a malware benchmark developed by the third party researchers. We also examine 18 real world malwares and show that the environment discrimination has not been widely employed by the malware developers.

7 Acknowledgement

The effort described in this article was partially sponsored by the U.S. Army Research Laboratory Cyber Security Collaborative Research Alliance under Contract Number W911NF-13-2-0045. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

1. Dave Chaffey. Mobile marketing statistics compilation. `http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/`. Accessed June 5, 2017.

2. Christian Lueg. 8,400 new android malware samples every day. https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day/. Accessed May 25, 2017.
3. Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.
4. Anand Lal Shimpi and Brian Klug. They're (almost) all dirty: The state of cheating in android benchmarks. http://www.anandtech.com/show/7384/state-of-cheating-in-android-benchmarks/. Accessed May 19, 2017.
5. Joel Hruska. Samsung goes legit, stops cheating on benchmarks with latest android update. http://www.extremetech.com/computing/177841-samsungs-latest-android-update-no-longer-cheats-on-benchmarks/. Accessed June 11, 2017.
6. Eric Mack. Htc admits boosting one m8 benchmarks; makes it a feature. http://www.cnet.com/news/is-the-htc-one-m8-that-good-benchmark-cheating-alleged-again/. Accessed June 10, 2017.
7. Russell Hotten. Volkswagen: The scandal explained. http://www.bbc.com/news/business-34324772/. Accessed June 4, 2017.
8. Wolfram Bartussek and David L Parnas. Using assertions about traces to write abstract specifications for software modules. In *Information Systems Methodology*, pages 211–236. Springer, 1978.
9. John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta informatica*, 10(1):27–52, 1978.
10. John McLean. A formal method for the abstract specification of software. *Journal of the ACM (JACM)*, 31(3):600–627, 1984.
11. James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
12. Lori A Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, pages 488–491. ACM, 1976.
13. Lok-Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX security symposium*, pages 569–584, 2012.
14. William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.
15. Golam Sarwar, Olivier Mehani, Roksana Boreli, and Mohamed Ali Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, pages 461–468, 2013.
16. Asia Slowinska and Herbert Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 61–74. ACM, 2009.
17. Lorenzo Cavallaro, Prateek Saxena, and R Sekar. On the limits of information flow techniques for malware analysis and containment. In *International conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163. Springer, 2008.
18. Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. Andrubis-1,000,000 apps later: A view on current android malware behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

19. P Lantz, A Desnos, and K Yang. Droidbox: Android application sandbox, 2012.
20. Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, (2):32–39, 2007.
21. Ulrich Bayer, Christopher Kruegel, and Engin Kirda. *TTAnalyze: A tool for analyzing malware*. na, 2006.
22. Norman safeground antivirus software. `http://www.norman.com/`. Accessed June 8, 2017.
23. Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.
24. Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013.
25. Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 417–426. ACM, 2010.
26. Joel Hruska. Android and security. `http://googlemobile.blogspot.it/2012/02/android-and-security.html/`. Accessed May 11, 2017.
27. Xuxian Jiang and Xinyuan Wang. out-of-the-box monitoring of vm-based high-interaction honeypots. In *Recent Advances in Intrusion Detection*, pages 198–218. Springer, 2007.
28. Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.
29. Prahlad Fogla and Wenke Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 59–68. ACM, 2006.
30. Boris Lau and Vanja Svajcer. Measuring virtual machine detection in malware using dsd tracer. *Journal in Computer Virology*, 6(3):181–195, 2010.
31. Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, volume 41, page 86, 2009.
32. Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.
33. Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 216–225. ACM, 2014.
34. Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: bare-metal analysis-based evasive malware detection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 287–301, 2014.
35. Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient detection of split personalities in malware. In *NDSS*, 2010.