Behavior Analysis via Execution Path Clustering

Rebecca Cathey
BAE Systems
Arlington, VA
rebecca.cathey@baesystems.com

Gregory Frazier
Apogee Research
McLean, VA
glfrazier@apogee-research.com

Michael Weber
BAE Systems
Arlington, VA
michael.weber@baesystems.com

***Abstract*—As the presence of malware increases in binary applications, behavior analysis is rapidly becoming necessary. We examine the application of execution path clustering and information pedigree analysis to analyze the behaviors of an application. An execution path is the sequence of basic blocks in a binary that are executed in response to a given input. One execution path represents a specific behavior of the application; likewise, similar execution paths define similar application behaviors. We cluster dynamic execution paths using the hierarchical agglomerative clustering algorithm to characterize program behavior. Furthermore, through comparisons between clusters, we can use information pedigree analysis to identify the modal inputs which cause the execution of unique behaviors within a cluster. Through this form of modality analysis, we can identify the modal inputs which control the mode in which the application executes. This approach allows us to automatically elicit the specification of software for which we only have the binary image. To assess the utility of this approach, we report on experiments conducted against a set of test Android applications.**

I. Introduction

In today's computing world, a user is often required to install opaque binary objects. Without these installations, certain desired behaviors, such as a browser correctly depicting content, would not be achieved. Another example is an application marketplace, such as GooglePlay [1] or Apple's App Store [2] which provide easy access to hundreds of thousands of mobile applications. It is generally assumed that the applications installed are safe and contain no hidden behaviors. These applications, however, at times contain either intentional or unintentional behaviors which have been known to leak sensitive information [3], download and install applications [4], or contain known malware [5]. In this paper, we present a technique for eliciting a human-readable description of the behavior(s) exhibited by a software binary, allowing one to determine whether there are hidden behaviors in the application that make it undesirable. Although our work is currently applied to Android applications, it is applicable for analysis of all types of binaries.

There are primarily two different ways to examine a binary application: static and dynamic. We focus on a dynamic approach to avoid limitations inherent to static analysis. We apply behavior analysis [6] to our dynamic analysis of a binary application.

We define an application's overall behavior as an aggregate of individual behaviors, where each *individual behavior* is a mapping of inputs to outputs that can be characterized algorithmically. Consider a calculator that supports addition, subtraction, multiplication and division—the calculator's behavior can be described as an aggregate of four individual behaviors, where each of those individual behaviors can be described by a function. Our goal is to offer a hierarchical decomposition of an application's behavior space that can be understood by an analyst. This understanding is assisted by identifying the *trigger* for each behavior cluster—the input that causes the behavior. We use the term *modal input* to characterize an input that changes an application's mode of operation—that triggers or elicits a new behavior.

A binary application can be represented as a set of basic blocks where each block has only one entry point and one exit point. The basic blocks are the vertices in an application's control flow graph (CFG). The directed edges of the CFG are the possible transitions from each basic block. By the definition of basic block, every basic block has more than one possible successor, and so more than one outgoing edge in the CFG.

Execution paths are sequences of basic blocks in the order they were executed. Execution paths can be represented as vectors of features created from any combination of basic blocks in an execution path. An execution path can be used to define a specific application's behavior. Grouping similar execution paths can then be used to group all observed behaviors into a subset of behaviors which cover an application's behavior space.

Information pedigree analysis determines the impact of an application's input on the actions performed. It is a form of data flow analysis that traces both implicit and explicit information flows. Looking at where the data is flowing for specific behaviors allows us to compare information flows between clusters to derive the specific inputs which affect the execution of the unique basic blocks.

We perform behavior analysis by instrumenting a binary application and running it in a customized execution environment. This allows us to inject inputs, observe the outputs, and capture the execution traces. Inputs are currently generated by using a fuzzer designed to exercise a significant portion of the code. We perform execution path clustering to derive classes of behavior. Applying information pedigree analysis over different classes of behavior allows us to perform modality analysis to find the find the inputs which affect the unique behaviors within a cluster and thus control a specific behavior, or the modal inputs which affect different modes of operation. The modal inputs can then be merged with our hierarchical view of all possible behaviors to allow an analyst to determine what the behaviors are and where they came from. Although,

The effort depicted is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL)

Approved for public release; distribution is unlimited. Cleared for open publication on 9/10/2013.

our hierarchical view of clusters does display behaviors to the analyst in the form of I/O flows, the focus of this paper is deriving the modal inputs which cause different classes of behavior to occur.

To summarize, the main contributions of this paper are:

- An execution path clustering approach to derive classes of behavior within a binary application and provide a hierarchical view of all observed behaviors.
- The combination of information pedigree analysis with our hierarchical depiction of an application's behavior space to elicit the modal inputs which cause the observed unique behaviors.

II. Related Work

Execution path clustering has achieved success in several domains. Execution path clustering has been effective for software optimization [7], [8], simulation [9], and finding failures [10]. In addition, the time varying behaviors of an application can also be used to improve simulation of application performance [11], [12]. In [7], repeated patterns of behavior are classified and used for optimizations. This work defines repeated patterns of behavior to be repeated executions of basic blocks. Characterizing the workload in this manner plays an important role in the design of efficient hardware and software systems. Prior work in execution path clustering defines a behavior as an execution profile. An execution profile is defined as executed system calls, basic blocks, or function callee/callers, We build on prior work, by applying execution path clustering to behavioral analysis.

Clustering has also been successfully used to classify malicious behavior in malware. Clustering of system calls, abstractions of system calls [13], or network traces [14] has been used to classify groups of malware to speed up the process of generating malware signatures. However, the focus is on classifying variants of the same malware based on the behaviors exhibited by malware. Our focus is on a lower level of granularity, we look at the behaviors within one application to give further understanding of an application's overall behavior.

Information pedigree analysis derives from two existing forms of program analysis. *Program slicing* [15], [16] is a static analysis technique for identifying the instructions that could affect the value of a variable at a particular point in the program. This technique has been extended to utilize dynamic analysis—dynamic program slicing [17] performs a similar function, but is able to generate a narrower slice. According to [18], dynamic slicing cannot on its own detect implicit information flows; it offers a static analysis step to convert them into explicit information flows. A second analysis technique is *taint analysis*, a form of data flow analysis in which one tracks the propagation of input state through an application. The general implementation is to associate a single bit of metadata with each memory location. When an input is read into a memory location, its bit is set. When multiple memory locations' values are used to set the value of another location, the destination location's bit is the logical OR of the metadata bits of the source locations. Dynamic taint analysis is described in [19], [20], and has been used in practical tools such as Valgrind [21] and TaintDroid [22].

III. Approach

A. Execution Path Clustering

An execution path consists of a trace, or sequence, of executed basic blocks. We assume a deterministic model of execution; any factor that might modify an execution sequence we define as *input*. Thus, the execution path is the behavior of the application for a given specific input. By this definition, any two inputs that elicit different execution paths represent different behaviors. *Execution path clustering* is the technique that we have applied to collecting execution paths into aggregate *behaviors*. Execution path clustering enhances code analysis by creating structure in the behavioral space. Thus, enabling us to identify and classify behaviors.

We use basic blocks to construct execution paths, similar to the method used in [7]. In our approach, an execution path is represented as a vector of features where each feature represents a unique basic block. At the end of execution, the number of times each basic block was executed is counted. Each element in the vector is mapped to a unique basic block where the element's value is how many times that basic block was entered during execution. Depending on the size of an application, a vector can be potentially large. However, since only a subset of the application is generally executed, the vector should be sparse. We weight each feature by the inverse vector frequency (ivf see (2)) to increase the weight of basic blocks which only appear in a small subset of the vectors. This emphasizes the unique behaviors.

We use the hierarchical agglomerative clustering algorithm to cluster execution paths. This algorithm runs in $O(n^2)$ time, where n is the number of execution paths. At the lowest level, the algorithm treats each vector as an individual cluster. The initial phase of the hierarchical agglomerative clustering algorithm is to calculate all vector to vector similarities. Then each vector is placed in a separate cluster. The clusters containing the two closest vectors are merged into a cluster during the first iteration of the algorithm. A new similarity measurement is then calculated between the new cluster and all other clusters. Subsequent iterations continue to merge the remaining clusters until only one cluster remains which contains all of the vectors. The result of the hierarchical clustering algorithm is a tree of clusters where each parent cluster contains two child clusters. When merging clusters, we use the Voorhees group average linkage criteria [23] to calculate the new similarities between the merged cluster and all other clusters as shown in (3). This criteria calculates the merged similarity as the average of the similarity metrics between the clusters, while taking into consideration the sizes of each cluster. We use the cosine similarity metric (see (1)) to calculate the initial vector to vector similarities. The cosine similarity metric iterates through all j basic blocks, Each feature is weighted by the number of times a basic block b appears in a specific vector i, $v_{i,b}$, multiplied by the inverse vector frequency ivf_b of that basic block. The inverse vector frequency of a specific basic block, j, is calculated as shown in (2), where n is the total number of vectors and n_j is the number of vectors containing j. Weighting features by the ivf reduces the impact of features which occur in a large number of vectors. This helps to control the fact that some basic blocks are more common than others. These basic blocks are often part of utility code and as such don't provide unique behaviors to reason over.

Utility code is the general purpose code that is useful in multiple runs of the same program. For example, in a Java application, portions of the main method are always called regardless of the input. Similarly, the code implementing a HashMap is called by all instances of a HashMap. Logic code, on the other hand, is the code that knits together the utility code in a way that implements logic specific to the behavior of an application. Using the ivf value to weight the features puts more weight on the logic code which will cause test cases with similar logic to group together. Logic code will manifest itself as unique segments of code that are not shared among the majority of the path vectors. Essentially, execution path clustering will weight features that have maximal impact on determining the unique behaviors of an application.

$$sim(v_1, v_2) = \frac{\sum^j (v_{1,j} * ivf_j) \cdot (v_{2,j} * ivf_j)}{\sqrt{\sum^j (v_{1,j} * ivf_j)^2}\sqrt{\sum^j (v_{2,j} * ivf_j)^2}} \quad (1)$$

$$ivf_j = log\left(\frac{n}{n_j + 1}\right) \quad (2)$$

$$sim(c_1, c_2) = \frac{\sum^j (c_{1,j} * \mid c_1 \mid) + (c_{2,j} * \mid c_2 \mid)}{\mid c_1 \mid + \mid c_2 \mid} \quad (3)$$

B. Information Pedigree Analysis

Like dynamic program slicing, Information Pedigree Analysis (IPA) constructs a graph of information dependencies that can be subsequently reasoned over, but both the nodes and the edges differ. The nodes of an IPA graph are *states* that are created during the execution of a program. At any given point in a program's execution, the program has a "value" which is the values stored in the registers and memory locations that comprise the program's state. We use the word *location* to describe any addressable state-holding element in the system, whether it is a register or memory. The state's *name* is the value that identifies the location. When a location is written to, a *state* has been created. In doing so, the previous value stored at that location, which was also a state, is overwritten and destroyed. We linearize every program execution such that every instruction execution is identified by a unique sequence number; the first instruction executed in the program has the sequence number one. Thus, at each sequence number, one or more states are read, their values are combined in some way, one or more new states are created, and the same number of states are destroyed. We identify a state by the duple $(name, seqNum)$ for the location name and the sequence number where it was created; the state is destroyed when a new state is created that has the same name and a greater sequence number.

The edges of the IPA graph link the states that were used to construct a new state to the that new state—they identify which states *taint* a given state. There are four classes of taint: *value*; *addr_rd*; *addr_wt*; and *control*. Figure 1 uses four code fragments to illustrate the four edge classes. In the first example, on sequence number τ_i a location named x is assigned the sum of the values in locations y and z. y and z received their values at some previous sequence numbers τ_a and τ_b, respectively, and those states are value-tainting x at τ_i. In the second example, x receives a value from some location m, where m is specified by y and i. Thus, y and i are addr_rd-tainting x—they specify the state that was read at τ_i. Similarly, in the third example, the location m that is value-tainted by y

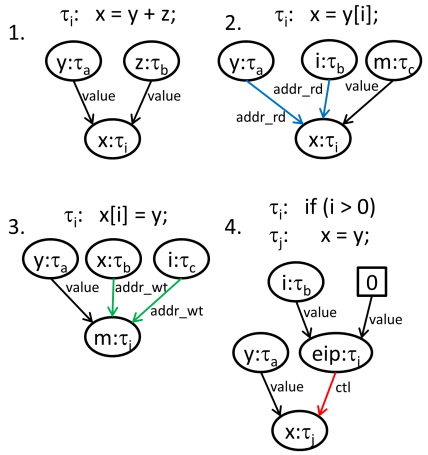

Fig. 1. Examples of the four classes of taint in an Information Pedigree Graph. In each example, a statement is executed at sequence number τ_i (and, in the fourth example, τ_j). Input states occured at sequence numbers τ_a, τ_b and τ_c. $\{\tau_a, \tau_b, \tau_c\} < \tau_i < \tau_j$. The location m in examples 2 and 3 is the memory referenced by $y[i]$ and $x[i]$, respectively.

at τ_i is specified by x and i, those states are addr_wt-tainting m. In the fourth example, there is a "hidden" state in the code, as the comparison "i $<$ 0" is an operation whose output state is stored in an unspecified variable. When operating in Java, we posit a per-thread location named "eip" in which branch-comparison state is held. Thus, in that example, eip receives value-taint from i and the constant state 0 at τ_i, and it in turn asserts control-taint on the creation of x at τ_j. In a simple binary language, such as Java bytecode, a single instruction will result in three to four taints with one or two destination states.

The information pedigree graph allows an analysis engine to traverse the graph backwards and forwards in time, understanding both directions.

C. Modality Analysis

As stated in the introduction, our goal is to elicit an application's behavior(s) in way that allows a human to understand the software. There are two elements to providing this understanding: describing the behavior space as a hierarchical clustering of similar behaviors and identifying the modal input that distinguishes one cluster from another. Our focus is on deriving the modal inputs, however, we display a simplistic view of the behaviors in the form of a cluster's I/O flows.

The result of execution path clustering is a tree of execution-trace clusters, with the set of all execution traces at the root. Each non-leaf node in the tree contains two children which represent the two clusters which were most similar at a particular iteration in the clustering algorithm. Since the features on which the clustering is based are the basic blocks that comprise a trace, what distinguishes the two clusters are the basic blocks that appear in the one but not the other. We describe these as the *distinguishing* blocks—the basic blocks that distinguish the one cluster from the other. The key insight is that something caused the execution traces in one cluster to execute the distinguishing blocks for that cluster, while the peer cluster in the hierarchy either bypassed those blocks or executed their own distinguishing blocks. There are two possibilities: either there is a basic block that precedes

the distinguishing block(s) that *are* in both clusters, or the distinguishing block(s) are the first blocks executed in the trace. We use the Information Pedigree graph to trace the distinguishing blocks to the inputs which caused them. Thus, we can identify the modal input(s) that control execution of those basic blocks.

IV. Evaluation

We developed prototype software, Droid Reasoning Analysis and Protection Engine (DRAPE), to provide automated capabilities to assist a human analyst to detect unwanted behaviors in Android mobile applications. DRAPE is a dynamic analysis tool which contains a customized dynamic execution environment and brings together concolic execution, information pedigree analysis and execution path clustering. We focus on the information pedigree analysis and execution path clustering components to show how they can be combined for modality analysis. We first show the utility of our approach through a detailed example. Then, we show the accuracy of our clustering approach. Finally, we report our initial results on the DARPA Automated Program Analysis for Cybersecurity (APAC) program.

Consider a four function calculator Android application with buttons for each number, `plus`, `minus`, `divide`, `multiply`, `equals`, and `C`. We modified this four function calculator to include a time component. The final calculator application is composed of approximately $27,000$ basic blocks. The average execution path length is approximately $3,400$ blocks and the average number of unique basic blocks between sibling clusters is 5. Whenever the time falls within a specified range, the app sends a file to an update website. This is malicious behavior embedded in the calculator, triggered by time. The code checking the time is only executed when the `equals` button is pressed. From DRAPE's point of view, time is an input, it is a trigger which causes a specific behavior. As each of these inputs may determine a unique path through the binary code, any of these buttons and the time has the potential to be a modal input.

We randomly generated 101 test cases using our four function calculator Android application. The test cases have varying operators (`plus`, `minus`, `multiply`, `divide`), number of operators, number of digits, presence of decimals, and times. Using DRAPE, we captured the execution traces for each test case, clustered the execution traces, and derived the modal inputs. The high level view of the modal inputs is shown in Fig. 2. In this figure, the number in parenthesis is the number of test cases grouped in that particular cluster. The values following the parenthesis are the modal inputs. For simplicity, we represent the modal inputs as the buttons pressed. In actuality, the modal inputs are the API calls which are treated as input (i.e., `performClick(){equals}`). These modal inputs are detected by looking at the inputs which caused path taint on the distinguishing basic blocks. Note that, the modal inputs do not necessarily show what behavior occurred in each cluster, rather they show what caused the unique behavior which occurs in each cluster. This is an important distinction as we further examine the modal inputs. DRAPE currently presents the behaviors as I/O flows of the API calls (i.e. `performClick() {equals} => setText{10}`). However, we focus on the inputs which cause the unique behavior within a cluster.

At the root node, we see all 101 test cases, This cluster is divided into two clusters, one containing 72 test cases and the other containing 29 test cases. The larger cluster contains no unique basic blocks when combined with the smaller cluster, thus no modal inputs at this level. The smaller cluster has two modal inputs: time and `divide`. This means that the time and the `divide` button both cause unique basic blocks to execute. This also means that the `divide` button is not pressed at all for the larger cluster. Thus, the unique behavior caused by the `divide` button only occurs in this smaller cluster. This cluster can be divided into several smaller clusters, one with 10 test cases and the other with 19 test cases. Note that the cluster with 10 test cases only contains the modal input of `C`. Furthermore, the sibling cluster has modal inputs of time, `plus`, `multiply`, and `minus`. Thus, the sibling cluster does not contain any unique behaviors which occur due to the `divide` button being pressed and we can safely assume that all behavior occurring because of the `divide` button occurs in both clusters. However, since the time is a modal input for the sibling cluster, the `divide` activity which occurs must occur during a specific time. Thus, although direct correlations between modal inputs and behavior cannot be made, indirect correlations based on which modal inputs appeared in the hierarchy above and on the same level as a particular cluster can be made. Furthermore, our software is eliciting the fact that time is an input. This should cause an analyst to wonder why time has any effect on the behavior of a calculator application.

The modal inputs we expected to see were the time and the four functions, as well as the `equals` and `C` buttons. Although, we saw these modal inputs, we found that the clusters occur first due to operation and secondly due to the `equals` and `C` buttons. This means that as the clustering algorithm runs, the clusters are first merged by operator, then the correlating `C` and `equals` cluster for a particular operation are clustered. Thus, the modal inputs showing up are more likely to be the `C` and `equals` button than the actual operator. Although, every cluster containing behavior performing the `plus` functionality occurs because of the `plus` button being pressed, the `plus` button may not show up as a modal input because it is also occurring in a sibling cluster. At some level, for some cluster, the `plus` button should show up as a modal input. The overall cluster containing 101 test cases shows 0 modal inputs, however, in actuality all basic blocks are unique, thus all inputs are path taints and thus modal inputs. Although trivial, this example illustrates how a hierarchy of clusters when combined with information pedigree analysis can elicit the modal inputs.

To test the accuracy of our execution path clustering algorithm, we performed a manual analysis. We compared levels of the cluster hierarchy with how a human would expect the clusters to divide. There are four functions which may be mixed with the `equals` or the `C` button. In addition, `equals` may also occur during a specified time. This makes twelve variations of the time, four functions, and termination (`equals` or `C`) buttons: `plus` combined with `equals` in each of two different time ranges, `minus` combined with `equals` in each of two different time ranges, `divide` com-

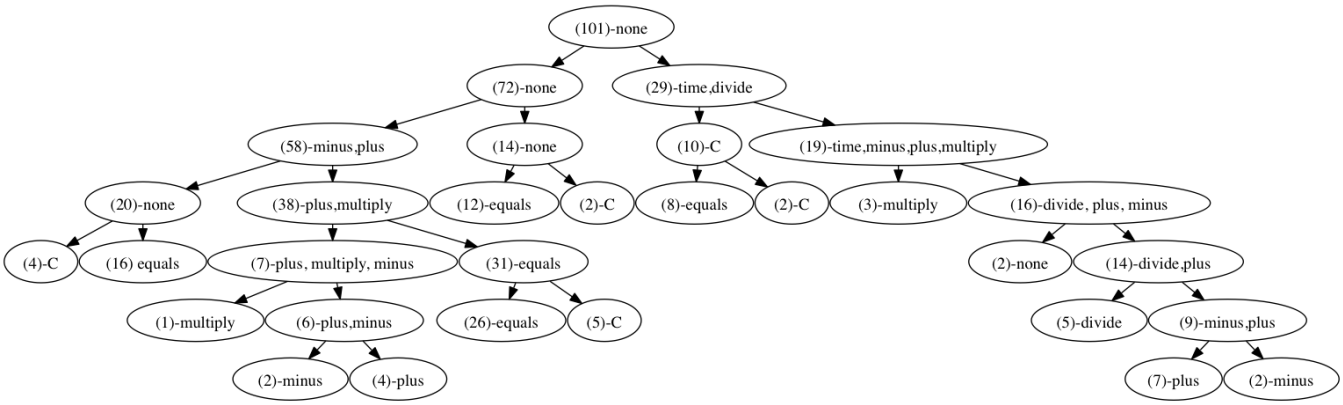

Fig. 2. Modal inputs for Calculator hierarchy of behaviors

bined with `equals` in each of the two different time ranges, and `multiply` combined with `equals` in each of the two different time ranges. The remaining four variations are the four operators combined with the `C` button. Thus, we calculate the accuracy of a given clustering against having twelve perfect clusters. As we merge from twelve clusters to one, we assume the perfect clustering will first merge clusters containing the `equals` and `C` for a given operator, then clusters of specific operand types regardless of the termination button. The time should be the final division. That is, when there are two clusters remaining, one should contain all the operators combined with a specified time, while the other should contain all of the operators where the control block within the time condition is not executed. The accuracy as the number of clusters increases is shown in Fig. 3. Accuracy is calculated using the number of false positives, true positives, false negatives, and true negatives. We calculate the accuracy on different levels of the clustering hierarchy. We examine levels containing between 76 and 2 clusters. We observe the best accuracy for eight clusters. For eight clusters, the accuracy is approximately 93%. For two clusters, the accuracy decreases by a small amount to 91%. This is because the `division` operator merges with the clusters based on time rather than with the clusters containing the `plus`, `minus`, and `multiply` operands.

As the number of clusters per hierarchy level decreases, we see an increase in the accuracy. Note that when we have twelve clusters, we do not see the exact division we expect. Neither did we see the expected division at eight clusters. This is primarily due to the number of operands in a test case. The number of operands causes a loop to be executed multiple times. This causes certain basic blocks to occur multiple times, giving them more weight. The number of operands had an unexpected impact. Although this feature does not impact the clustering based on operation, it does affect the accuracy based on our assumptions. Several ways to improve this include taking into consideration loops, or using a binary metric for feature weight instead of the frequency combined with the inverse vector frequency.

Finally, we report our results from DARPA′s Automated Program Analysis for Cybersecurity (APAC) program. We were given a set of Android applications containing both benign and malicious functionality. For the applications examined, we systematically explored all, or a majority, of the programs execution paths. We then clustered the resulting execution traces to derive classes of behavior. Viewing the

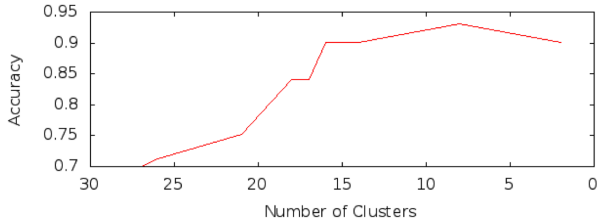

Fig. 3. Accuracy of Clustering Behaviors

hierarchy of behaviors allowed us to find the modal inputs causing each class of behavior. Due to unsupported APIs, we were unable to run our system on the entire set of test apps, however, we did run our system successfully on a smaller subset and correctly determined the modal inputs for these applications. The modal inputs were derived with a combination of execution path clustering combined with information pedigree analysis. Following are some examples of a few of the apps examined and how we were able to elicit the modal inputs for the malicious activity.

App-1 is a GPS application which can perform three actions: (1) the application receives the GPS coordinates from the Android device, but nothing occurs, (2) the GPS coordinates are received which allows a button to be clicked, or (3) the GPS coordinates are never received causing no activity to occur. We ran App-1 with a set of test cases exercising these three actions and then clustered the execution paths. In one cluster, we show that the unique behavior is triggered by calling the `MyLocationOverlay.getMyLocation` API call. An analyst could look deeper into the application by examining the input to output flows (derived via information pedigree analysis) within the cluster to determine that an output of the cluster is an exec call which is a ping to a specified IP address. Furthermore, the ping command has a pattern specified by the `p` flag which may be related to the location coordinates. Thus, the `MyLocation.getMyLocation` API call is a trigger for a ping.

A second application, App-2, had three main buttons which a user could interact with: A view button, a backup button, and a delete button. We created test cases to exercise these three buttons and ran them. Then clustered the resulting test cases. Our clusters showed that not only were the buttons the modal inputs, but that extra inputs were being used when the backup button was pressed. In particular, this cluster shows a modal input as an `ApplicationContentResolver` query with the value of `content://sms`, As

well as a `managedQuery` with the value of `content://browser/bookmarks`. Finally, there is also an input from `java.io.File.lastModified` which occurs on the images in the Camera images directory. Thus, the backup button, bookmark data, image data, and SMS data are all modal inputs for the behavior which occurs when the backup button is pressed, the inputs from the bookmarks, images, and sms messages are used. The analyst was then able to determine that this particular application should not be able to see images or bookmark data, thus, it was performing malicious behavior.

In a third application, App-3, which notifies a user when an SMS message which matches a specified filter arrives. For this application, we observed two primary classes of behavior: one which occurred when an SMS message is received and the other when an SMS message is sent. The first behavior was triggered by the `onReceive` method and the second was triggered by the `sendBroadCast` method.

Although initial, these results show that using information pedigree analysis combined with execution path clustering is a powerful approach to deriving those inputs which affect unique behaviors.

V. Conclusions

We present a technique for classifying the observed behaviors of a binary application and deriving the modal inputs which caused those behaviors. Our execution path clustering approach provides a hierarchical view of all observable behaviors. We combine our hierarchical depiction of the behavioral space with information pedigree analysis to trace unique behaviors back to the modal inputs which caused those behaviors to occur. This combination of execution path clustering and information pedigree analysis can be used to elicit a human-readable description to the observed behaviors exhibited by a software binary. Deriving the modal inputs is the first step in automatically eliciting a human-readable specification of an application's suite of behaviors as it allows us to map which inputs caused unique behaviors. The second part of eliciting a human-readable specification is to identify those unique behaviors. Future work will identify and present the unique behaviors to the user in the form of a specification. This specification would allow the analyst to fast and effectively evaluate an Android applications.

We developed software to extract behaviors from Android applications. We showed how our software was able to successfully cluster and extract the modal inputs for a four function calculator app. Furthermore, we were able to successfully elicit the modal inputs from a subset of Android test applications as part of DARPA's APAC program. Our results show the promise of this form of behavioral analysis.

Acknowledgment

Special thanks to Mike Ter Louw and Tavaris Thomas from LGS Innovations and Marc Krull from BAE Systems for their work on the dynamic execution environment.

References

[1] "Google play," https://play.google.com/store.

[2] "The app store," http://www.apple.com/iphone/from-the-app-store/.

[3] G. Keizer, "Free android apps scrape personal data, send it to china," http://www.computerworld.com/s/article/9179894/Free_Android_apps_scrape_personal_data_send_it_to_China, 2010.

[4] Lookout, "Update: Security alert: Hacked websites serve suspicious android apps (notcompatible)," https://blog.lookout.com/blog/2012/05/02/security-alert-hacked-websites-serve-suspicious-android-apps-noncompatible/, 2012.

[5] K. Mahaffeyt, "Security alert: Droiddream malware found in official android market," https://blog.lookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/, 2012.

[6] N. Provos and T. Holz, *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley Professional, 2007.

[7] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *10th International Conference on Architectural Support for Programming*, 2002.

[8] E. Duesterwald, C. Cascaval, and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003.

[9] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2003.

[10] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," in *the 23rd International Conference on Software Engineering (ICSE '01)*, 2001.

[11] G. Hamerly, E. Perelman, J. Lau, B. Calder, and T. Sherwood, "Using machine learning to guide architecture simulation," *Journal of Machine Learning Research*, vol. 7, pp. 343–378, 2006.

[12] E. Perelman, J. Lau, H. Patil, A. Jaleel, G. Hamerly, and B. Calder, "Cross binary simulation points," in *International Conference on Performance Analysis of Systems and Software*, 2007.

[13] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Network and Distributed System Security Symposium (NDSS*, 2009.

[14] R. Perdisci, W. Lee, and N. Feamster, "Behavioral clustering of http-based malware and signature generation using malicious network traces," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[15] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, pp. 446–452, July 1982.

[16] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 1, pp. 26–60, January 1990.

[17] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, 1990.

[18] W. A. Masri, "Cynamic information flow analysis, slicing and profiling," Ph.D. dissertation, Case Western Reserve University, January 2005.

[19] J. Newsome and D. Song, "Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software," in *Network and Distributed Systems Security Symposium*, February 2005.

[20] E. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *IEEE Symposium on Security and Privacy*, May 2010.

[21] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *ACM Conference on Programming Language Design and Implementation*, 2007, pp. 89–96.

[22] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10, 2010.

[23] E. M. Voorhees, "Implementing agglomerative hierarchic clustering algorithms for use in document retrieval," *Information Processing and Management*, vol. 22, no. 6, pp. 465–476, December 1986.