Improving PDG Vector Creation for AnDarwin

Julia Matsieva • ECS 235A • Fall 2012

UC Davis

Abstract

The Android application market allows developers to create original and innovative applications using the Android platform; however, malicious developers may try to profit off advertising revenue by stealing code from existing applications. Others may release cloned applications in order to increase their chances of being downloaded. In order to assess and monitor the health of the Android market, the AnDarwin project aims to detect these duplicate applications in a scalable way. This is done by constructing dependence graphs of Android applications, converting them into vectors and identifying similar vectors via efficient clustering methods. Here, we suggest and evaluate several proposals to improve the creation of PDG vectors that retain more information about graph structure, helping distinguish applications that are falsely marked as clones using the current approach.

1. Background

The popularity of smart phones has jumped significantly in the past few years and a large component of the Android experience is the third-party marketplaces, where developers can create their own applications and original content for use on the Android platform. However, due to the infrastructure in place for profiting off these application, there is some incentive for malicious developers to steal existing apps and pass them off as their own in order to steal ad revenue. Similarly, some developers will often release duplicates of the same application in order to get the attention of users. These practices potentially clutter the market and make it difficult for honest developers to create and market their high-quality product. Thus, the AnDarwin project is motivated by the desire to identify spam or plagiarism that hurt the Android market ecosystem.

The goal of AnDarwin is to identify applications that share a significant amount of code without relying on metadata, thereby enabling it to identify both cloned and plagiarized applications. To accomplish this, AnDarwin relies on *program dependence graphs* which are constructed from the DEX code of each method of an Android application as follows: each statement s in the code becomes a node in the graph and there is an edge (s, t) between every pair of statements s and t if t contains a variable whose value depends on s. Thus, the edges represent data dependencies rather than control flow dependencies. Each graph is then converted to a set of *semantic vectors*, with each vector corresponding to a connected component of the graph. The d-dimensional vector v is constructed by setting the value of v_i, the i^{th} component of the vector, to be the number of nodes of type i that appear in the connected component, where d is the number of statement types. For example, if a connected component contains two conditional branches and they have type $i = 9$, then the value of v_9 will be 2. These resulting sets of vectors are then clustered using Locality Sensitive Hashing, which efficiently identifies vectors that are close together in Euclidean space.

This approach allows AnDarwin to achieve major advantages in scalability and efficiency. In order to accurately compare two Android programs, it would be necessary to detect whether two PDG's have the same structure; however, this would require solving the *maximum common subgraph isomorphism* problem, which is known to be NP-hard. Thus, AnDarwin simplifies the task by performing the vector conversion step and comparing the resulting vectors instead. Similarly, AnDarwin leverages the LSH algorithm in order to avoid pairwise comparisons between vectors, which would be quadratic in the number of vectors and thus prohibitive for analyzing libraries of over 300,000 Android applications. [1]

2. Problem

As shown in the previous section, AnDarwin makes impressive gains in efficiency and scalability by performing the conversion between PDG and semantic vectors. However, from the construction given, we can see that the conversion from a PDG to its corresponding set of vectors does not retain *any* edge information — important existing knowledge that could potentially increase the accuracy of vector comparison. Furthermore, the authors of [1] state that

> To improve the scalability of this approach further, we partition the semantic vectors into overlapping partitions based on the vector magnitudes. The intuition behind this step is that connected components of significantly different size are unlikely to be clones.

However, it is often in the nature of security research to begin an arms race and now that this optimization is known, it presents a vulnerability in the AnDarwin methodology. Android spammers and plagiarists may attempt to take advantage of the lack of edge information in semantic vectors by inflating the sizes of PDG components, and maybe even combining connected components through bogus data dependencies.

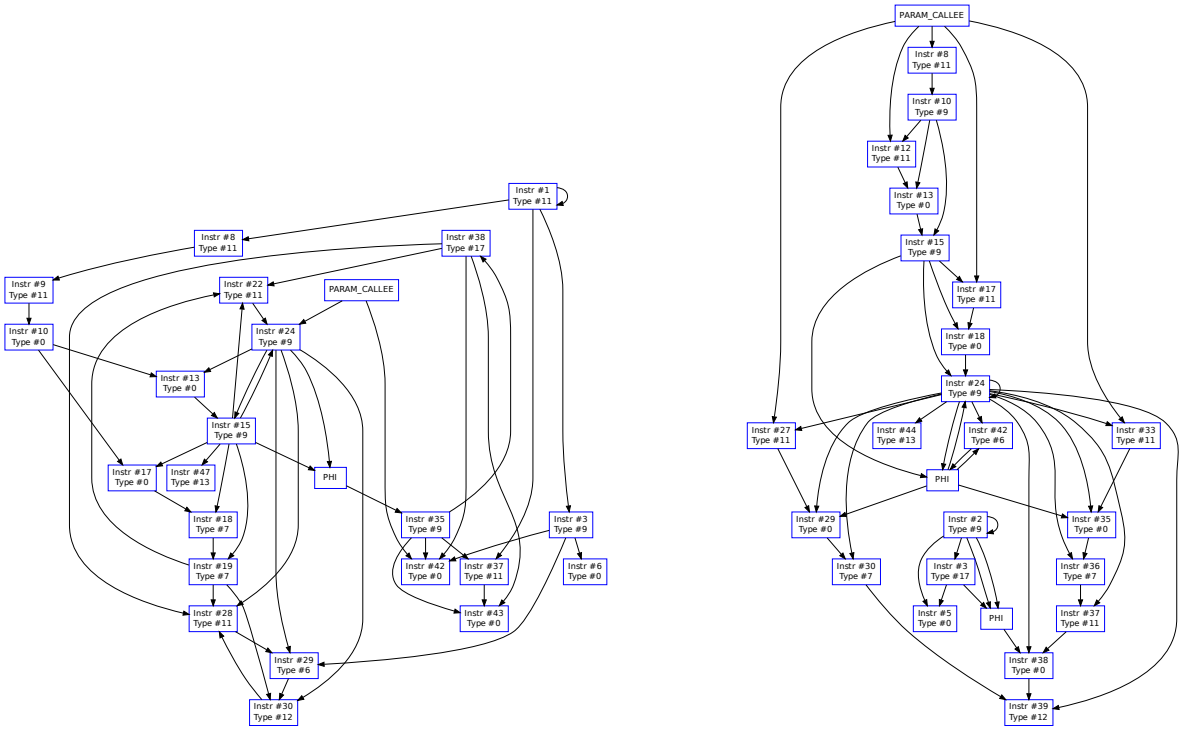
Therefore, we seek to improve the PDG-to-vector conversion process in a way that incorporates more structural information from the original graph. Furthermore, we wish to find solutions that:

- do not introduce any new false negatives
- are somewhat resistant to simple obfuscation techniques
- do not significantly inflate the runtime of AnDarwin.

In the sections that follow, we propose and evaluate several ways of incorporating characteristic graph edge information into semantic vectors.

3. Simple degree counts

When considering a method to condense the edge information of a graph into a vector, one may be tempted to express the entire graph as a vector in the form of an adjacency matrix or an adjacency list. However, since connected components may have different sizes, this will create vectors of different dimensions that cannot be compared



(a) PDG 1 (b) PDG 2

Figure 1. Non-isomorphic connected components that share the same vector under the current construction

easily. Furthermore, for dense graphs $|E| \in O(|V|^2)$, which means the dimension of the vectors to be clustered will potentially jump as the square of number of nodes. Finally, we know that the LHS algorithm has running time [1]

$$O(d \sum_{g \in G} |g|^\rho \log |g|),$$

and vector dimension d is currently kept constant by categorizing statement types. If the entire graph is encoded as a vector, the vector dimension will no longer be a constant. Therefore, encoding the entire graph as a vector is undesirable.

Meanwhile, a simple and promising to condense characteristic structural information of graph into a vector is to count the degrees of nodes of each statement type and record them in the vector. This can be done through aggregation or computing the maximum or the average degree for each node type. Also, since PDG's are directed graphs, the number of edges coming *into* a node, or its *in-degree*, can be recorded separately from its *out-degree*. Figure 1 shows a false positive – two PDG's that are structurally different but have the same vector under the current construction in AnDarwin. It is easy to see that, even though the two graphs have the same number of nodes of Type 9, these nodes have very different degree corounts.

Since the set of nodes of a certain type and their edges induce a subgraph on the PDG, there are several different ways to characterize the d subgraphs using degree counts. Yet, regardless of which method is chosen, we can show that any method already has two out of the three desired properties — it will not generate any new false negatives and will not considerably inflate the runtime of the LHS algorithm. The first property is easy to see: if two graphs are isomorphic, then they have identical structure. Therefore, all of their nodes will have the same degree.

[1] G is the set of vector partitions, $|g|$ is the size of the vector partition and $0 < \rho < 1$

The second property comes from the fact that the measure of degree is computed for each of d node types, so even if c different degree counts are recorded in the vector, this will only increase the vector dimension, and therefore the runtime of LSH, by a constant factor of c. Finally, these degree measures can be computed in time $O(|V|+|E|)$ in the size of the PDG, which increases the complexity of vector creation somewhat compared to the previous construction, which did not look at edge data at all.

Therefore, these methods will only provide different benefits regarding resistance to obfuscation. These are evaluated below.

3.1 Total degree

In the spirit of the original vector construction used by AnDarwin, we can modify vectors to record the *sum* of the degrees of all the nodes of each type. That is, if V_i is the subset of nodes of type i in a program dependence graph G, then set

$$v_{d+i} = \sum_{u \in V_i} \deg(u).$$

However, this construction increases the sensitivity of vectors to small changes in graphs. For example, consider the graph G and construct a graph G' by adding a path $u_1, \ldots u_k$ of nodes of some type to some node $u \in G$. This change might correspond to a string of bogus a linear chain of bogus variables that depend on each other, and is a simple obfuscation that can be made by an attacker. Under the old construction, v'_i constructed from G' would be k greater than v_i. Meanwhile, under the new construction, the v'_i would also increase by k, but v'_{d+i} would increase by $2k$, thereby increasing the distance between v and v' even more than under the old construction, even though the graphs were structurally similar except for that one simple obfuscation. Therefore, this method is not recommended due to its sensitivity to small changes.

3.2 Max degree

A less precise but also less sensitive measure is the *maximum* degree of a node type.

$$v_{d+i} = \max_{u \in V_i} \deg(u).$$

For example, we can see in Figure 1(a) that the two central nodes of Type 9 have out-degree 5 and 6, while the central node of Type 9 in 1(b) has out-degree 12. In a way, max degree records the relevance of the nodes that are most *important*, thereby providing useful structural characterization, but not in a way that is too sensitive to obfuscation. Under this construction, the change described in the previous section will affect the original component v_i in the same way, but v_{d+i} will increase by at most 1, if at all. This approach will also bridge some distance between structurally similar graphs with different node counts.

3.3 Max in-degree vs. max out-degree

Since PDG's are directed, it may help to separate the in-degree count from the out-degree to provide even more structural information. Furthermore, it may even be more useful to record one or the other in the vector, as they have different meanings in the PDG and therefore respond differently to obfuscation attempts. More specifically, the in-degree of a statement s measures how many statements the value of s depend on, which is difficult to modify because when something s depends on something, it is difficult to *force* it to depend on something else. However, it is easy to create additional meaningless statements with high in-degree that do not mean anything but depend on a lot of things. Meanwhile, the out-degree of a statement s measures its importance to the rest of the program, so it may be easy to create bogus statements and make them also depend on s to drive up its out-degree. However, the creation of a new statement s' that has higher out-degree than s is fairly difficult without creating many additional statements, since it is difficult to *force* existing values to depend on s'.

Therefore, this presents a trade-off. When an attacker tries to increase v_i when it is computed as the maximum in-degree, the attacker will likely opt to create a bogus variable and have it depend on all the values in the program. Meanwhile, if v_i is the maximum out-degree of a statement of type i, then an attacker will have trouble beating v_i by creating a new statement of high out-degree, so he will attempt to add bogus statements that depend on the statement s_M of type i with max out-degree. However, we argue that since s_M is already the most important statement in the program, the attacker will be more reluctant to add extra bogus statements and create complexity to an already important statement than he would in the case of obfuscating for in-degree. Therefore, we believe that the out-degree measure is somewhat more obfuscation-resistant.

3.4 Combined approach

However, it is possible to combine the trade-offs in the two approaches above. We claimed that existing in-degree is difficult to modify and existing max out-degree is difficult to beat with a new statement. Therefore, we propose the following construction:

$$v_{d+i} = \deg_{in}(\arg\max_{u \in V_i} \deg_{out}(u_i)).$$

The rationale for this is as follows: the statement of type i with the max out-degree can be considered the *most important* statement of this type. We argued that if an attacker was to modify the program, he would increase the max out-degree of this statement rather than add a new statement of higher out-degree. Similarly, we argued that the in-degree of an existing statement is difficult to change. Therefore, we suggest identifying the most important statement in the program and then recording its in-degree.

3.5 Average in- or out-degree

Finally, it is possible that many of these vector modifications will create many new false positives. This is because no characterization of the set of PDG's is currently available, so it is very likely that these graphs are not randomly distributed. Indeed, these graphs are creates from similar programs, so it is likely that they are structurally similar in ways we did not anticipate and thus *some* of these degree measures result in very close vectors for programs that are otherwise very different. For example, it might turn out that the max out-degree of all addition statements is always 2 because of the way addition is used in programs. We do not know that this is not the case. Therefore, in this situation, we suggest adding the measure of *average* in-degree or out-degree to further distinguish programs that may inadvertently turn out to be similar under this method.

Additionally, most simple obfuscations would likely not involve building many connections to the existing program (since they do not want to accidentally break it.) Therefore, it is likely that most obfuscation would build weakly connected chains of bogus nodes – which may drive up max degree measures but would not significantly drive up the average.

3.6 Conclusion

As an initial modification to AnDarwin, it appears that the following construction would be the simplest and most promising

$$v_{d+i} = \max_{u \in V_i} \deg_{out}(u),$$

since it captures the importance of a program statement rather than its data dependencies. However, since no concise characterization of the data set exists, it is possible that all Android programs are rather structurally similar or contain many similar methods, and therefore it is impossible to say whether this approach will generate a lot of *new* false positives. If this is the case, then it may be helpful to combine max out-degree with other degree measures of the graph such as average out-degree to better characterize structure.

While we argued that the combined approach is most resistant to obfuscation, it is likely that it is unnecessary at this time as not many developers will be trying to obfuscate that aggressively. Meanwhile, this technique is *more* likely to generate more new false positives, since it's fairly likely that certain important statements will have similar in-degree. For example, most conditional branches may have in-degree two or three because they do not involve many variables. Therefore, this construction is only recommended in conjunction with the technique shown above and will likely only perform well after attackers get creative.

4. Future work

The first step in future work would obviously be the implementation and testing of these techniques. However, it is not entirely sufficient to see how well they do in practice; it is also desirable to obtain some information about when they are performing well, when they fail and which obfuscations are effective at confusing them. Therefore, we believe that this project would benefit from investing in an automated way to characterize false positives. This would likely be done by identifying the positives, verifying them with a graph isomorphism solver and then summarizing where the algorithm went wrong. This data could then be used to evaluate and advance vector construction methods to further refine the AnDarwin tool.

Another useful modification to the AnDarwin approach may be to distort Euclidean space to account for some inequalities in the axes. For example, suppose type i is binary operations and type j is conditional statements. It may be *easier* for an attacker to add extra binary operations to a program than to add conditional statement. Or, it could be that the range of average degree counts is much smaller than the range of the max degree counts. Therefore, it may

help to lengthen that axis where on which points naturally cluster close together or compress the axes on which points naturally lie far apart. However, since there is no ground truth, this is difficult to accomplish correctly but it may be possible with a representative random sample of the data and some statistical analysis.

References

[1] Jonathan Crussell, Clint Gibler, and Hao Chen.
Attack of the Clones: Detecting Cloned Applications on Android Markets.
To appear in *17th European Symposium on Research in Computer Security (ESORICS)*, Pisa, Italy, September 10-12, 2012. (20%)