Systematic Audit of Third-Party Android Phones

Michael Mitchell, Guanyu Tian, Zhi Wang
Florida State University
{mitchell, tian, zwang}@cs.fsu.edu

ABSTRACT

Android has become the leading smartphone platform with hundreds of devices from various manufacturers available on the market today. All these phones closely resemble each other with similar hardware and software features. Manufacturers must therefore customize the official Android system to differentiate their devices. Unfortunately, such heavily customization by third-party manufacturers often leads to serious vulnerabilities that do *not* exist in the official Android system. In this paper, we propose a comparative approach to systematically audit software in third-party phones by comparing them side-by-side to the official system. Specifically, we first retrieve pre-loaded apps and libraries from the phone and build a matching base system from the Android open source project repository. We then compare corresponding apps and libraries for potential vulnerabilities. To facilitate this process, we have designed and implemented DexDiff, a system that can pinpoint fine structural differences between two Android binaries and also present the changes in their surrounding contexts. Our experiments show that DexDiff is efficient and scalable. For example, it spends less than two and half minutes to process two 16.5MB (in total) files. DexDiff is also able to reveal a new vulnerability and details of the invasive CIQ mobile intelligence software.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Information flow controls; D.2.5 [**Software Engineering**]: Testing and Debugging—*Code inspections and walk-throughs*

Keywords

Android; DexDiff; BinDiff; Security Audit; Static Analysis

1. INTRODUCTION

Recent years have witnessed the increasing adoption of smartphones. According to a recent report by Gartner [22], there were more than 150 million smartphones sold to end users in the second quarter of 2012, an increase of 42.7% year over year. Android-based smartphones lead the market share with nearly 99 million (64.1%) units sold, surpassing the second market leader (iOS) by 45.3%. The vast popularity of Android can be partially attributed to the wide variety of Android-based smartphones from many manufacturers (or vendors) such as Samsung, LG, and HTC. Currently, there are more than 210 Android smartphones from 24 well-known manufacturers being sold worldwide. Among them, 75 Android phones from 16 manufacturers are available in the US market as of September 2013 [25]. Many of these phones closely resemble each other as they all follow the same hardware design guideline and run the similar Android-based software. This lack of product differentiation may lead to competitive disadvantages. As such, third-party manufacturers heavily customize the official Android system to differentiate their products from others. Major Android manufacturers all have their own distinct flavors of Android such as HTC Sense and Samsung TouchWiz.

Unfortunately, deep customization by third-party manufacturers often introduces vulnerabilities that do *not* exist in the official Android system. For example, the HTCLoggers application [50] in many HTC phones was found to collect lots of sensitive information, and provide it over a local network port accessible to any application with the INTERNET permission. Also, researchers found that many Android phones sold by the major US operators are pre-installed with "rootkit-like" software from CarrierIQ [13], which can remotely collect "a vast array of metrics" including the received calls and locations, posing serious threats to user privacy. Moreover, design flaws vulnerable to the confused deputy attack [32] were identified in pre-installed apps. These flaws can be leveraged to indirectly break the Android permission model [19, 29]. In light of these serious vulnerabilities, there is a pressing need to systematically audit the third-party customization in these commercial off-the-shelf Android smartphones.

In this paper, we propose to audit third-party Android phones by comparing them side-by-side to the official Android system based on the key observation that design flaws often creep in through the vendor customization [13, 29, 50]. This comparative approach allows us to quickly locate the manufacturer's modifications to the original system and further assess their security impacts. Specifically, we first need to obtain a copy of the pre-installed apps and libraries from the phone, and build a matching system from the Android open-source project [26] (for brevity, we call it "the base"), then compare corresponding apps and libraries from the phone being assessed with and the base for potential vulnerabilities. To facilitate this process, we built DexDiff, a tool that can automatically pinpoint fine differences between two Android binaries. Although Android consists of both Java and native code that may be customized by a vendor, DexDiff focuses on the Java-based Android binaries. Previous systems target only native binaries [20,21]. Thus, with the help of DexDiff, an external security analyst, who

CODASPY'14, March 3–5, 2014, San Antonio, Texas, USA.

ACM 978-1-4503-2278-2/14/03 ...$15.00.
http://dx.doi.org/10.1145/2557547.2557558.

has no access to the source code, can focus their efforts on modifications where vulnerabilities are most likely to be introduced, thereby significantly reducing the time and efforts needed to audit a phone.

However, providing such functionality is a challenge. First, manufacturers almost always keep user-space apps and libraries close-sourced, as permitted by their liberal licenses (core Android libraries and apps usually are licensed with the three clause BSD or the Apache license, unlike the Linux kernel which has a GPL license.) Therefore, it is not feasible to directly use existing source code diffing tools such as the UNIX `diff`. Instead, we need to directly compare Android binaries. Second, it is also infeasible to simply disassemble the binaries [1] and then compare the resulting assembly code with `diff` or a similar tool because the compiler uses complex algorithms to achieve an optimal layout for the binary. Possibly, even simple insertion or deletion of instructions can lead to dramatic changes to the instruction layout. It is thus necessary to structurally compare these two binaries (i.e., to compare their control flow graphs [3]). Structural comparison also brings the benefit of putting these modifications into the surrounding context to help the analyst assess their impacts. Third, the solution should be able to scale to the size of commodity Android binaries, which could contain tens of thousands of methods. For example, the `framework.dex` file in our test phone consists of 5,423 classes and 52,566 methods with a file size of 9.9MB, while its corresponding base binary has 3,924 classes and 38,283 methods in a 6.6MB file. Large files pose significant challenge to DexDiff because algorithms for graph comparison usually have limited scalability. By applying a series optimization, DexDiff can handle the aforementioned files in about two and half minutes.

To address these challenges, we design a tool called DexDiff that can pinpoint fine differences between two Android binaries. Unlike other tools [11, 58] [1] that linearly compare disassembled Android instructions, DexDiff structurally compares the control flow graphs (CFG) [3] of two Android apps. Given two Android binaries, DexDiff works in two phases. In the first parse, it parses the binaries into classes and methods, and uses a fast but coarse-grained similarity comparison algorithm to find an assignment of classes and methods (one from each file), in which each assigned class or method is sufficiently similar to its peer; In the second phase, DexDiff constructs the control flow graphs of each pair of matched methods, and performs a fine-grained graph-based comparison of their CFGs using an approximation of the maximum common sub-graph isomorphism problem [2]. By doing so, we can heavily reduce the workload for the more expensive graph-based comparison, thus making DexDiff scalable to commodity Android binaries. We have implemented a prototype of DexDiff and used it to audit a popular Android phone to demonstrate the effectiveness of our approach. Through examining the manufacturer's modifications of the Android framework, we discovered a vulnerable device management app that accepts commands from a local network port without authenticating the connection first. This particular app is loaded with Android permissions, and willingly provides private information obtained through these permissions to any app with the Internet access permission. Moreover, we verified that the phone has the CarrierIQ software embedded [13]. With the help of DexDiff, we further systematically report metrics collected by the software, especially in the stock Android browser. To summarize, this paper makes the following contributions: ,

[1] Our tool is not related to [58], which is incomplete and inactive. Both projects were started at roughly the same time.

- Based on the observation that vulnerabilities often are introduced by vendor customization, we propose an approach to systematically audit third-party Android phones by comparing them side-to-side against the official Android system.
- To facilitate the audit of third-party phones, we have designed DexDiff, a tool that can pinpoint modifications to Android binaries and highlight these changes in their surrounding context, allowing security analysts to focus their efforts to locations that are most likely to cause problems. Although each individual algorithm of DexDiff has been applied in security before, the overall design of DexDiff is efficient and scalable.
- We have implemented a prototype of DexDiff. Our evaluation shows that DexDiff is efficient and scalable. It is also effective in leading to the discovery of new vulnerabilities in our test phone.

The rest of this paper is organized as follows: we first describe the design and implementation of DexDiff in Section 2, and then evaluate the system performance and effectiveness in Section 3. Afterward, we discuss possible limitations and future improvements in Section 4, and present related work in Section 5. Finally, we conclude the paper in Section 6.

2. DEXDIFF DESIGN

2.1 Overview

DexDiff is designed to help (external) security analysts to assess security impacts of vendor customization to the official Android system. It structurally compares two Android binaries and highlights the changes in their surrounding context, i.e., the control flow graphs [3]. To use DexDiff, the analyst first obtains a copy of the pre-install Android apps and libraries from the phone, and builds their corresponding base binaries from the release branch in AOSP [26] on which the phone is based. Then, the analyst uses DexDiff to compare each pair of the Android binaries and evaluate the security impacts of individual modifications. Although the security evaluation still requires human expertise (probably no automated systems can eliminate such a need), DexDiff can significantly reduce the time and effort required by directing them to locations where problems are most likely to occur. To this end, we have three design goals for DexDiff:

- *Accuracy*: DexDiff should accurately locate changes to an Android binary and present them in the right context. In particular, it should be able to tolerate common mechanical changes, for example, to the instruction layout by an optimizer, or to the class names that are often targeted by trivial obfuscators [43]. The former requires a non-linear comparison (e.g., graph-based), while the latter prevents any exact matching.
- *Scalability*: Our solution should scale to commodity Android binaries, which could consist of hundreds of thousands methods even for pre-loaded apps and libraries. However, algorithms for graph-based comparison (i.e., isomorphism) usually have high resource demands. Therefore, it is necessary to pre-process the binaries with a faster (but less precise) algorithm to reduce the workload for graph comparison.
- *Efficiency*: The third requirement is closely related the second one. Our solution should not only scale to large inputs, but also process them quickly because, as an aid to the analyst, DexDiff will be used interactively.

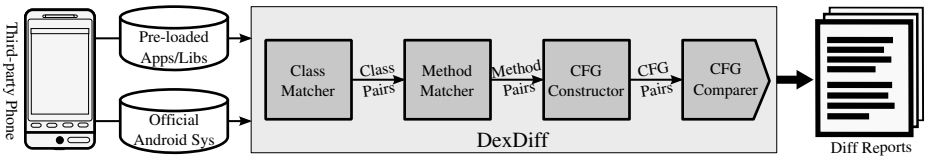

Figure 1: Overview of DexDiff

Figure 1 shows a high level overview of DexDiff. Given two Android binary files, DexDiff produces a diff report in two phases. In the first phase, DexDiff uses a fast but coarse-grained similarity algorithm to quickly match classes and methods. Specifically, it parses the binary files into Java classes, and uses code similarity to find for each class its match in the peer binary, if there is any (*Class Matcher*). For every pair of matched classes, DexDiff applies the same technique to find for each method its match in the peer class (*Method Matcher*). In the second phase, DexDiff leverages a slower but more precise algorithm to compare the control-flow graphs for each pair of matched methods. Specifically, it breaks the pair of matched methods into basic blocks and constructs the control flow graph for each method (*CFG Constructor*). It further uses a graph isomorphism engine to find a maximum matching of these basic blocks. The resulting matched basic blocks consist of unchanged or modified basic blocks, while unmatched basic blocks are basic blocks that have been deleted or inserted, depending on which file they belong to (*CFG Comparer*). To facilitate the auditor, DexDiff lays out the CFGs of each pair of matched methods side-by-side in a graph (using dot [6]), and visually highlights their differences. Figure 3 shows an example output of DexDiff. In the following subsections, we will give more details of each step.

2.2 Matching Classes

We rephrase class matching as an assignment problem [55]: given two disjoint set of classes from two Android binaries, DexDiff tries to find an optimal assignment of classes in which each class is assigned to a resembling class in the other file (if there is any). To solve this problem, DexDiff first parses the input files into classes, and performs a pair-wise comparison of these classes (one from each input file) to calculate their similarity. This generates a matrix of similarity scores. It then leverages the *Hungarian algorithm* [57] to find an optimal assignment of these classes that maximizes the overall similarity between the two files.

Parsing Android Binary Given two Android binaries, DexDiff first parses them into classes. Android's Dalvik class file has a different format than the standard Java class file: the definition for each class in the standard Java is contained in a separate file; while in Android, a single class file (the `classes.dex` file in an `apk` file) includes definitions for all the classes of an app or library. For example, `framework.dex`, the Android framework file, in the official Gingerbread firmware (Android version 2.3.3) contains 3,924 classes, while the same file in the HTC's Gingerbread firmware for HTC EVO 4G has 5,423 classes, or a 38.2% increase over the official firmware. Encapsulating all classes in a single file allows Android to reduce the binary size by maximizing shared identifiers and values.

Calculating Similarity After parsing the inputs into classes, DexDiff performs a pair-wise comparison of classes (one from each input) to calculate their similarity. This amounts to $p \times q$ comparisons if the inputs have p and q classes, respectively. For example, it would requires 21,279,852 comparisons for the two previously mentioned `framework.dex` files. It is thus critical to construct an efficient algorithm for each comparison. To this end, DexDiff first converts each class into its string representation and further leverages feature-hashing [56] to reduce the overhead of n-gram based similarity.

Specifically, DexDiff computes class similarity in two steps. It first converts each class into its string representation for easy similarity calculation. The string representation of a class is a concatenation of its name, super class name, implemented interfaces, class and member fields, and methods. Particularly, a method consists of its signature (i.e., parameters, return value, and exceptions it may throw), name, and disassembled instructions. To tolerate differences in compiler's register allocation, DexDiff ignores registers in instructions but leaves other references intact. This is an improvement over previous system designs that rely (almost) solely on opcodes [31, 59] because rich type information is encoded in the Dalvik bytecode and retained in DexDiff. For example, the `new-array vA, vB, type@CCCC` instruction [27] constructs a new array of size `vB` and puts the result into register `vA`. `type@CCCC` is a reference to a type in the symbol table. If only opcode is considered, this instruction will be abbreviated as `new-array`, losing the type information. In DexDiff, we resolve all the references such as types, method prototypes, and strings, and append them to the opcode of the instructions. For instance, the `new-array v0, v1 [Ljava/lang/String` instruction that creates a string array will be reduced to `new-array [Ljava/lang/String` in DexDiff. Similarly, method invocation instructions in DexDiff (i.e., `invoke-kind`) keep the method prototypes to be called, such as the `invoke-virtual Lcom/android/browser/WebDialog-> findViewById(I)Landroid/view/View` instruction. It is clear from these examples that disassembled instructions in DexDiff keep rich semantic information about the original program.

After converting classes into strings, DexDiff uses n-gram to calculate their similarity. Previous research [37, 38] has shown that n-gram based similarity is reliable against reordering, insertion, and deletion. To measure similarity between classes, DexDiff slides a window of length n over the strings for the class, advancing the window one string at a time. Each n-gram of strings is considered as a unit for comparison. We then calculate the similarity as the percentage of common n-grams out of the total unique n-grams. That is, the similarity of two classes c_a and c_b can be expressed as the Jaccard index of:

$$Similarity(c_a, c_b) = \frac{|c_a \cap c_b|}{|c_a \cup c_b|} \quad (1)$$

As mentioned earlier, DexDiff performs a pair-wise comparison of classes to generate a p-by-q matrix of similarity scores. However, string comparison used by n-gram similarity is an expensive operation, especially when implemented naively. For example, the number of string comparison operations if we compute Equation (1) by pair-wise string comparison is: $\sum_{\substack{1 \le i \le p \\ 1 \le j \le q}} |c_i| \times |c_j|$ which is prohibitively expensive for large Android binaries. To improve the efficiency in calculating similarity, we adopt an algorithm called

feature hashing, which has been shown to closely approximate the true similarity if the hash function uniformly distributes its output in a large value space [38]. More specifically, we compute the hash (a 32-bit number) for each n-gram, and replace the n-grams with their hashes in Equation (1). As such, set operations in strings in Equation (1) are substituted by set operations in integers. However, unlike Juxtapp [31] and ReDeBug [37] that encode hash values in a large bit-vector (e.g., $128K$ bits), we directly use dynamic arrays to save hash values. This design is chosen because n-gram sets for many Android classes are very sparse if encoded in bit-vectors. Performing $2p \times q$ set operations on these sparse bit-vectors (as required by pair-wise similarity comparison) is prohibitively time-consuming. Therefore, in DexDiff, we store the hashes in a dynamic array, and normalize the array by sorting it and removing duplicates before set operations. The set operations thus can be efficiently implemented as the intersection and merging of two sorted arrays.

Matching Classes Given the matrix of similarity scores calculated in the previous step, we can rephrase class matching as the assignment problem [55]. Specifically, we can view classes in one input file as workers, and classes in the other input file as tasks. Each similarity score can then be interpreted as the gain for a worker to complete a task. The assignment problem looks for an optimal assignment of the workers to the tasks so that the total gain is maximized. The Hungarian algorithm [57] is a well-known algorithm to solve the assignment problem in polynomial time. In DexDiff, we apply this algorithm to find an optimal assignment of classes that maximizes the overall similarity for the input files. The outputs of this step are pairs of assigned (or matched) classes.

2.3 Matching Methods

Given a pair of matched classes, DexDiff further refines the result by finding an assignment of methods between each pair of classes. The approach is the same as that used to match classes. More specifically, DexDiff first converts these methods into their string representations, and calculates a matrix of similarity scores between each pair of the methods using n-gram based similarity. It then finds an optimal assignment of methods that maximizes the overall similarity for these two classes. The output of this step is pairs of matched methods, which will be further processed to structurally compare their differences.

2.4 Constructing Control Flow Graphs

DexDiff compares two Android binaries in two phases. In the first phase, it uses a faster but coarse-grained algorithm to find a matching between classes and methods. Each pair of matched methods will be further processed in the second phase to reveal their structural differences. Specifically, DexDiff first constructs the control-flow graph for each method, and then uses an approximation algorithm for the maximum common subgraph isomorphism problem [2] to structurally compare these two graphs. The resulting subgraph contains basic blocks shared between these two methods, while an unmatched basic block is either deleted or newly inserted, depending on the graph to which it belongs.

DexDiff uses a traditional algorithm [3] to construct CFG for a method. It first goes through all the instructions to identify basic blocks. A basic block is a straight-line sequence of instructions with only one entry and one exit. Any instruction that may change control flow is an exit, including unconditional jump (e.g., `goto`), conditional branch (e.g., `if-eqz`), method invocation (e.g., `invoke-virtual`) and return, as well as any instructions that may throw an exception. However, a large number of Dalvik instructions [27] may cause exceptions (103 out of 246 defined bytecode, or 41.9%), including some frequently used ones such as `iget` and `iput` to access a field in an object, and `aget` and `aput` to access an array. If we allow these instructions to terminate a basic block, the average size of basic blocks becomes too small, usually containing one or two instructions. This will increase the overhead of our graph isomorphism engine because there are many more basic blocks to consider and more basic blocks are similar to each other. Therefore, we do not terminate a basic block with this kind of instructions unless it also belongs to an earlier type such as method invocation or it is the `throw` instruction. After identifying basic blocks, DexDiff goes through the list of basic blocks and connects them together to form the CFG for the method.

2.5 Comparing Control Flow Graphs

DexDiff compares CFGs of each pair of matched methods to locate code modifications, i.e., which basic blocks have been changed, inserted, or deleted. This problem can be rephrased as the *maximum common subgraph isomorphism* problem, which seeks to find the largest common isomorphic subgraph of two input graphs. Each node (basic block) in the resulting common subgraph is shared by these methods and its counterpart in the other method is also given. Meanwhile, a node outside of the common subgraph is not matched, i.e., the basic block is inserted or deleted. More formally, given two graphs $\mathcal{G}_1 = (V_1, E_1)$ and $\mathcal{G}_2 = (V_2, E_2)$, their maximum common subgraph is the largest subgraph $\mathcal{H}_1 = (\bar{V}_1, \bar{E}_1)$ of $\mathcal{G}_1$ that is isomorphic to a subgraph $\mathcal{H}_2 = (\bar{V}_2, \bar{E}_2)$ of $\mathcal{G}_2$. Therefore, subgraph $\mathcal{H}_1$ and $\mathcal{H}_2$ are of equal size and satisfy the structural requirements of isomorphism. That is, there is a bijective mapping from $\bar{V}_1$ to $\bar{V}_2$ that preserves the connectivity between these nodes.

In applications of the maximum common subgraph problem, there are often additional requirements regarding nodes and edges. In DexDiff, the inputs to the problem are two CFGs with basic blocks being their nodes. The format and semantics of basic blocks can vary significantly from one to another. DexDiff thus allows only similar basic blocks to be matched. Two basic blocks BB_1 and BB_2 are considered to be similar if $Similarity(BB_1, BB_2) > \theta$, in which θ is a configurable threshold (DexDiff uses 80% for θ). DexDiff uses the same n-gram based similarity for basic blocks. However, it is not feasible to use the Hungarian algorithm to match basic blocks (like what we have done in matching classes and methods) because many basic blocks are exactly the same or very similar to each other. It is necessary to use the connectivity between basic blocks to get an accurate result. Meanwhile, edges in a CFG represent control flow between basic blocks. They can be categorized according to their types of control flow, such as conditional jump, unconditional jump, fall through, and return [3]. However, it might be too restrictive to require matched edges in the common subgraph to have the same type because it is common to change control flow type, for example, by adding a conditional test. Therefore, DexDiff does not impose additional requirements on edges.

Maximum common subgraph isomorphism is a NP-complete problem [2]. However, efficient approximation algorithms exist for two similar graphs. In DexDiff, the class and method matching ensure that two CFGs to be compared reasonably reassemble each other. We adopt such an algorithm called backtracking [42]. Intuitively, the backtracking algorithm explores all possible assignments of nodes for the input graphs. Its efficiency lies in aggressively pruning dead branches during the exploration. This algorithm fits for comparison of CFGs, in which most nodes have a relatively small number of incoming and outgoing edges. Backtracking has been applied before to binary diffing for native programs [21]. We refine the same algorithm for Android apps.

```
1:  procedure Backtrack(𝒟)
2:      if Extendable(𝒟) then
3:          v = PickNode(𝒟)
4:          𝒵 = GetMappableNodes(v, 𝒟)
5:          for all w ∈ 𝒵 do
6:              ℳ = ℳ + {(v, w)}
7:              𝒟′ = Refine(𝒟)
8:              Backtrack(𝒟′)
9:              ℳ = ℳ − {(v, w)}
10:         end for
11:         𝒱 = 𝒱 − {v}
12:         Backtrack(𝒟)
13:         𝒱 = 𝒱 + {v}
14:     else if |ℳ| > |ℛ| then
15:         ℛ = ℳ
16:     end if
17: end procedure
```

Figure 2: Backtracking algorithm

The backtracking algorithm (Figure 2) provides a framework to solve the subgraph isomorphism problem by enumerating all possible assignments. The inputs to this algorithm, CFG $\mathcal{G}_1$ and CFG $\mathcal{G}_2$, are global variables (as well as $\mathcal{V}$, $\mathcal{M}$, and $\mathcal{R}$). $\mathcal{V}$ is the node set of $\mathcal{G}_1$. Set $\mathcal{D}$ contains all the pairs of basic blocks that remain to be matched. $\mathcal{D}$ initially consists of all the basic block pairs that satisfy the similarity requirement, i.e., $Similarity(BB_1, BB_2) > \theta$. Set $\mathcal{M}$ contains node pairs that have already been matched so far. The algorithm first checks whether $\mathcal{M}$ can still be extended by $\mathcal{D}$ (line 2). If not (line 14-16), $\mathcal{M}$ is maximal and thus a candidate for the maximum common subgraph. It is then compared to $\mathcal{R}$, the temporary result so far (Line 14), and replace it if $\mathcal{M}$ is larger (Line 15). When `Backtrack` returns, $\mathcal{R}$ contains the final result. If $\mathcal{M}$ is extendable (Line 3-13), `Backtrack` performs a depth-first search for larger subgraphs (Line 3 to 10). Specifically, it first selects a still unmatched node v (Line 3) from $\mathcal{G}_1$, and gets the set of nodes in $\mathcal{G}_2$ that can be matched to v (Line 4), i.e., nodes that are similar to v and do not violate the structural requirement of graph isomorphism. Next, it performs a depth-first search (Line 6 to 9) for each possible match (Line 5): in Line 6, it extends $\mathcal{M}$ by (v, w). However, adding (v, w) to $\mathcal{M}$ may render some potential matches in $\mathcal{D}$ invalid. For example, any matches involving v or w are no longer necessary and should be removed. The pruning of invalid matches in $\mathcal{D}$ is performed at Line 7. In the next Line, `Backtrack` recursively extends $\mathcal{M}$. This recursion returns when $\mathcal{M}$ cannot be further extended (Line 2). Line 9 restores $\mathcal{M}$ for the next potential match of v. Moreover, it is possible that the maximum common subgraph might not contain v at all. This possibility is explored in Line 11 and 12 by temporarily removing node v from consideration, and recursively computing a subgraph without v. Line 13 returns v to $\mathcal{G}_1$'s node set for future consideration.

The backtracking algorithm in Figure 2 provides a framework to solve the maximum common subgraph problem by exhaustive search. Its efficiency (or even feasibility) depends on how aggressively dead branches can be trimmed. Each application of the algorithm needs to implement as effectively as possible for these four sub-routines: `Extendable`, `PickNode`, `GetMappableNodes`, and `Refine`. (1) DexDiff returns `true` for `Extendable` if there are still unmapped nodes, and it is possible to find a larger subgraph than the current one ($\mathcal{R}$) with the remaining nodes. In other words, DexDiff stops searching the branch as soon as there are not enough nodes for $\mathcal{M}$ to have more nodes than $\mathcal{R}$. (2) It picks the node

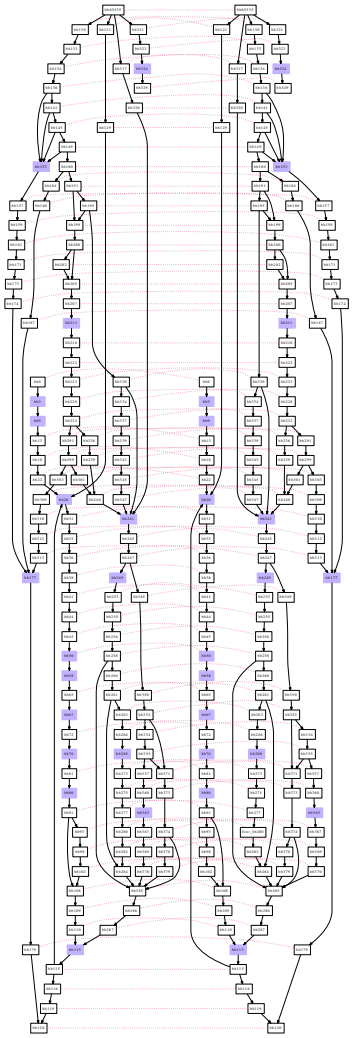

Figure 3: DexDiff example output

with the highest similarity score and the smallest number of candidates in `PickNode`. It is beneficial to try good matches first because `Extendable` will reject a larger number of smaller-sized matching faster. (3) `GetMappableNodes` returns all the nodes in $\mathcal{G}_2$ that are similar to node v and satisfy the structural requirements. These nodes are ordered by their similarity to node v for the same reason to try good matches first; (4) `Refine` prunes set $\mathcal{D}$ by removing any candidates that involve v or w because v and w have already been matched. It also removes any candidates that are structurally incompatible with node v and w. Because CFG is a directed graph, we require structural compatibility in both incoming and outgoing directions. More precisely, we remove any match (n_1, n_2) having

$$e(v, n_1) \neq e(w, n_2) \quad or \quad e(n_1, v) \neq e(n_2, w)$$

$e(a, b)$ is a function returning whether edge (a, b) exists in their graph. For example, $e(v, n_1)$ returns `true` if edge (v, n_1) exists in graph $\mathcal{G}_1$ and `false` otherwise.

By eagerly pruning dead branches, the backtracking algorithm converges quickly for most inputs. However, there still exist cases that require too much time or memory to complete. To address this problem, we introduce timeout into the algorithm to limit the total number and depth of the recursive calls. The timeout is only acti-

	Item	Configuration/Version
Desktop	System	Dell Optiplex 9010
	Distribution	Ubuntu 12.04 LTS 64-bit
	Kernel	Linux 3.2.0-33 x86_64 SMP
	graphviz (dot)	2.26.3
Phone	Model	HTC Evo 4G (Sprint)
	Firmware	4.54.651.1
	Android	2.3.3
	Radio	2.15.00.0808
	Kernel	Linux 2.6.35.10-g13578ee

Table 1: Configuration for the evaluation

vated after the algorithm has already found at least one assignment (i.e., $\mathcal{R} \neq \emptyset$). With these customization and optimization of the algorithm, DexDiff can handle all the tested binaries in less than three minutes.

2.6 Prototype

We have built a prototype of DexDiff in the C programming language. The prototype has about 13,600 lines of source code in total, of which about two third (9,200 SLOC) is a library to parse the Dex file format [27]. Each component of DexDiff (Figure 1) is multi-threaded. For example, class matcher calculates a matrix of similarity scores for two binaries. It first parses the binary into classes and further converts them into hashes of strings. It then uses multiple threads to calculate the pair-wise similarity between these classes with each thread handling roughly equal number of the class pairs. Other components are similarly multi-threaded.

To facilitate the analyst, outputs of DexDiff are visualized using "dot" [6] to highlight differences between methods and their surrounding context. Figure 3 shows an example output of DexDiff. In this figure, each rectangle represents a basic block, labeled with the index of its first instruction [2]. Matched basic blocks are linked by dotted (red) lines. Filled (blue) basic blocks are different for the two methods. Given the size of the methods (much larger samples exist in our test), it would be highly difficult and error-prone to manual locate these changes, let alone putting them into the correct context.

3. EVALUATION

DexDiff uses an approximation algorithm to solve the NP-complete maximum common subgraph isomorphism problem (Section 2.5). In this section, we first measure the performance of DexDiff to demonstrate its practicality, and then evaluate its effectiveness by systematically studying HTC EVO 4G [34] (for brevity, we will refer to this phone as "the phone" subsequently in this section.)

3.1 Performance Evaluation

To measure the performance of DexDiff, we extracted the sixteen Android files under the `/system/framework` directory of the phone and built their corresponding counterparts from the Android open source project repository. We use the Linux `time` command to measure how long it takes for DexDiff to process each pair of these binaries. We repeat each experiment three times and report the average time. The variations between these executions are negligible as the approach is deterministic except for the scheduling of multi-threads. All the experiments use eight threads, the same number as the CPU threads. As shown in Table 1, all the experiments were conducted on a desktop machine with a 3.4G Hz quad-core Intel Core i7-3770 CPU and 16GB memory. The processor has four (real) cores and each core has two hyper-threads [36], which are treated by Linux as real CPU cores. The results are summarized in Table 2.

[2]Due to the space constraint, simplified labels are used in Figure 3. The actual labels also include the complete disassembled instructions of the basic block.

policy:	2	javax:	1
core:	3	service:	12
framework:	38	sqlite-jdbc:	1

Table 3: Number of timeouts

These sixteen pairs of Android binaries cover a wide range of file sizes (from a few kilo-bytes to about ten mega-bytes) and numbers of classes (from one to more than five thousands). The execution time accordingly varies from less than 0.01 seconds to about 160 seconds. Notice that the time used by DexDiff is not directly correlated to the binary file size. For example, it takes DexDiff 7.5 times longer to handle `framework.dex` than `core.dex` even though the former is only about 30% larger than the latter. This uncertainty is caused by the backtracking algorithm [42], which exhaustively searches for the maximum common isomorphic subgraphs (the n-gram string similarity algorithm and the Hungarian algorithm [57] are both polynomial.) Execution time of backtracking depends heavily on how effective it can eliminate dead branches. To quantify its impact, we run the experiments twice with and without graph comparison. The latter only calculates similarity between two binaries, while the former also compares their structural differences. Their difference roughly equals to the time consumed by graph comparison. The results are shown in Figure 4. For experiments like `android.policy.dex` and `service.dex` that have relatively small input files but long processing time, graph comparison monopolies the execution time at around 99%. Also, graph comparison for `core.dex` takes only a short period because its methods are relatively small and, more importantly, do not have complicated control flows. The last column of Figure 4 shows the average percentages of similarity calculation (29%) and graph comparison (71%).

As mentioned earlier, we set a timeout on the backtracking algorithm (Section 2.5) to limit the total number of recursive calls. We empirically choose one million as the threshold. Larger thresholds fail to effectively reduce timeouts, but also considerably increase the process time. The number of timeouts provides an indication to the relative complexity of the corresponding CFG. For example, `service.dex` timeouts 12 times during graph comparison, and in total it requires about 160 seconds to complete. Table 3 lists the number of timeouts for all the tests with timeouts. We further examine these 38 cases in `framework.dex`. The leading causes of timeouts are: (1) large methods with many basic blocks that do not belong to the common subgraph, i.e., many new or deleted basic blocks. These basic blocks may cause discontinuity in the partial result ($\mathcal{M}$), leading the algorithm to hunt for the next continuous part of the subgraph. (2) many identical basic blocks. Some methods repeatedly use the same instruction sequence to perform a series of similar operations (e.g., to initialize a `HashMap`). These instruction sequences become identical after we remove register references. The algorithm slows down because the possible matches returned by `GetMappableNodes` contain many equally good choices. (3) nodes with large ingress and egress degrees. In CFG, most basic blocks have 1 or 2 outgoing edges determined by the exit instructions (fall-through, unconditional, or

Name	Base			Phone			Modified		New		Time
	Size	Class#	Method#	Size	Class#	Method#	Class#	Method#	Class#	Method#	
am	25KB	6	38	26K	6	39	1	1	0	1	0.10
policy	179KB	96	833	258K	140	1,200	29	105	44	368	10.23
test.runner	172KB	105	1,001	172K	105	1,002	2	1	0	1	0.17
bmgr	12KB	2	25	14K	2	27	1	2	0	2	0.05
bouncycastle	678KB	507	3,186	677K	507	3,186	31	47	0	0	1.37
location	6.3KB	4	56	6.3K	4	56	0	0	0	0	0.01
core	4.1MB	3,009	27,952	4.1M	3,017	28,093	353	1264	8	141	17.04
core-junit	21KB	19	142	21K	19	142	0	0	0	0	0.05
ext	1.2MB	960	6,896	1.2M	960	6896	209	468	0	0	40.01
framework	6.6MB	3,924	38,283	9.9M	5,423	52,566	1,198	4,556	1,504	14,290	144.70
ime	5.6KB	1	10	5.6K	1	10	0	0	0	0	0.03
input	3.6KB	1	7	3.6K	1	7	0	0	0	0	0.01
javax	53KB	24	164	53K	24	164	2	9	0	0	6.55
monkey	79KB	50	237	76K	50	237	1	1	0	0	0.17
pm	25KB	7	43	25K	7	43	1	2	0	0	0.09
services	1.3MB	437	4,014	1.7M	531	5,153	124	384	95	1,139	159.86
sqlite-jdbc	130KB	29	858	130K	29	858	3	16	0	0	2.49
svc	7KB	6	26	7.3K	6	26	1	1	0	0	0.03

Table 2: Processing time for files under /system/framework. Note that: (1)the size columns specify the size of the uncompressed `classes.dex` file in the apps. (2) The time reported here is the wall clock time (in seconds) to execute the command. (3) As expected, almost no classes or method are removed from the base.

conditional) except the `switch` instruction. However, the number of incoming edges to a basic block is not limited. To make it even worse, the proceeding (or succeeding) nodes to this kind of nodes are likely similar to each other. For example, exception handlers tender to start with a `move-exception` instruction followed by a `goto` instruction to jump to a common handler. The backtracking algorithm cannot quickly converge because its dead branch pruning (Line 7) is ineffective in this case.

In addition to limit numbers of recursion, we also constrain its depth to prevent memory depletion because extra memory is allocated for each call. This rule only affects one method during our test, `org.ccil.cowan.tagsoup.HTMLSchema.<init>`) in `ext.dex`, which uses 2,859 sequential basic blocks to initialize the HTML schema. Without this rule, the method causes DexDiff to nest too many times because DexDiff performs depth-first search and eventually depletes the memory. We did not encounter any other cases that cause the problem.

3.2 Effectiveness Evaluation

In this section, we demonstrate the effectiveness of our method by reporting the results of auditing the phone. During our audit, we systematically compare the common Android framework files (under `/system/framework`) and pre-loaded apps (under `/system/app`) of the phone and its base. The framework should be carefully audited because many of its libraries are loaded into every Dalvik virtual machine and thus accessible to all the apps. Moreover, APIs for propriety services provided by the vendor likely are exposed in one of the framework libraries as well. Table 2 summarizes the vendor's modifications to the framework. Even though these files are about 33 MB in total, one of the authors spent *less than 5 days* to process all of them because DexDiff allows analysts to focus on changes presented in their surrounding context.

Using DexDiff, we find that the vendor's modifications to the framework concentrate on five binaries: `android.policy.dex` enforces the device security policies such as mandatory screen lock. The vendor customizes this file to change the look and feel of the screen lock and to support its proprietary touch-based stylus; `core.dex` contains the core Java language public APIs and other popular Java libraries such as the Apache Harmony library [5]. DexDiff shows that changes to this file are minor and sporadic. Most of these changes seem to come from Google rather than introduced by the vendor. This is possible because DexDiff builds the base according to the Android version reported by the phone. Therefore, our base is likely to be slightly different from the actual base. This difference introduces noise into the system. In Section 4, we will discuss methods to locate or closely approximate the actual base; Changes to `ext.dex` have the same nature as `core.dex`. A few new Java libraries are added to `ext.dex`, such as the open-source NIST SIP stack, Apache logger, and the Apache HTTP library. The vendor extensively customizes `framework.dex` (the core library of the framework), and `services.dex` (the host of a number of services such as window manager and activity manager). For example, it adds the 4G and WiMax wireless network support, a proprietary logging facility [50], HTC Pen support, USB-based networking, HDMI and Bluetooth support etc to these files. In the following of this section, we report a number of representative unsafe customization revealed by DexDiff.

Case Study 1: Broadcast Input Services `framework.dex` is the core library of the Android framework (also the largest one). It is loaded into every Android app, and thus accessible to all the running apps. With DexDiff, we find that three new APIs have been added to `android.view.IWindowManager`. Their names indicate that they can enable broadcast of input events of the keyboard, touch screen, and trackball:

```
void broadcastKeyinEvent    (boolean);
void broadcastMotionEvent   (boolean);
void broadcastTrackballEvent(boolean);
```

`android.view.IWindowManager` defines a Java interface to access the Android window manager [24] defined in `services.dex`. At runtime, all the services in `services.dex` are hosted in the `system_server` process thus unreachable to apps directly. Instead, apps need to use inter-process communication (IPC) to access these services. The corresponding implemen-

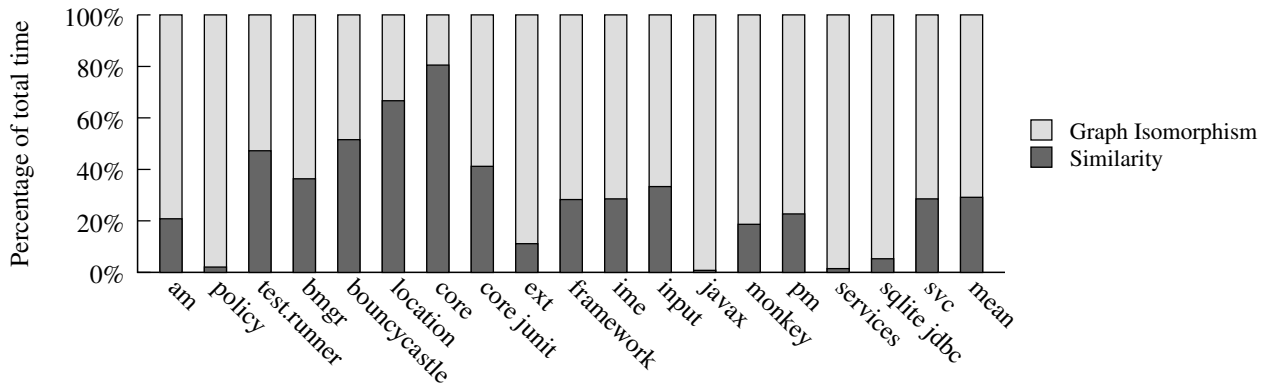

Figure 4: Distribution of processing time between class/method matching and CFG comparison

tation of the `IWindowManager` interface uses IPC to remotely invoke these three methods. Following the leads, DexDiff reveals that three new methods are added to `com.android.server.WindowManagerService` in `services.dex` that will actually enable the broadcast of these input events. If these methods are not guarded by proper permissions, it is a serious security issue since any apps can log user inputs. In the following, we describe how we try to exploit these potentially "rewarding" services and how it leads to the discovery of another vulnerability.

Since these APIs are a part of `framework.dex` and accessible to normal apps, we first try to call them directly. However, these APIs are adequately protected by the vendor-specific permissions, for example, the `com.htc.Manifest.permission.BROADCAST_KEYIN_EVENT` permission is required to call `broadcastKeyinEvent`. In addition, these permissions have the "SIGNATURE" protection level, and thus can only be granted to apps signed by the vendor. Normal apps cannot be granted these permissions. Again, following the leads, we try to audit pre-loaded apps that have these permissions. In this phone, these permissions are only granted to the `com.htc.android.omadm.service` package (`/system/app/HtcDm.apk`), a proprietary app without a counterpart in the base. Further inspection shows that this app has a plethora of dangerous permissions, for example, `INSTALL_PACKAGES` and `DELETE_PACKAGES` to install and delete apps, `MASTER_CLEAR` to factory-reset the phone, and `HTC_FOTA_UPDATE` to update the firmware. It can also access the Internet with its `INTERNET` permission. We further use DexDiff to generate its CFGs to study its internal. Supposedly, the app is designed to perform remote device management tasks such as wiping the phone in case it is lost or stolen. As such, it can read and execute commands from the IPC interface or the Internet [3]. However, the app has *no* authentication at all on the input. It blindly accepts and executes any commands received from these interfaces. Fortunately (and somewhat ironically), the most dangerous commands such as remotely bricking the phone have not been completely implemented yet. When triggered by our test app, these commands write a log to `logcat` and return an `OK` status. Nevertheless, we are able to retrieve some private information about the phone such as its device ID and software configuration by exploiting this app. Our inspection of the app shows that the input broadcast services (e.g., `broadcastKeyinEvent`) are used by the app to intercept user inputs during the firmware update. With the help of DexDiff, we are able to quickly identify this *new* vulnerability.

Case Study 2: CarrierIQ CarrierIQ is a piece of monitoring software tightly integrated into many smartphones. It can be instructed by the manufacturer or the service provider to gather, store, and retrieve diagnostic information (also call metrics) about the phone. The software can provide valuable information to the service provider about user experiences and software reliability. However, the vast array of metrics collected by the software leads to serious privacy concerns [13].

Our test phone is also loaded with the CarrierIQ software. The CarrierIQ agent resides in the `com.htc.android.iqrd` package (`/system/app/IQRD.apk`). With the help of DexDiff, we find that the base Android system has been extensively modified by the vendor to insert numerous hooks which will notify the agent of interested events. The captured metrics are forwarded to the agent through targeted intents. The phone defines 41 such intents, whose names follow the common pattern of `com.htc.android.iqagent.action.*`. For example, there are 11 UI-related metrics named such as `com.htc.android.iqagent.action.ui01`. The metrics collected by the software are thus meticulously categorized and fine-grained.

The vendor has strategically implanted hooks for the agent all over the phone to capture "a vast array of metrics". As mentioned earlier, the Android framework libraries are loaded into every app or manage data for them(e.g., `services.dex`). Inserting hooks in these libraries allows the vendor to log events for both pre-loaded apps and apps downloaded from the online app store. For example, in `services.dex`, DexDiff uncovers a hook inserted into the `reportLocation` method of `GpsLocationProvider` that can log detailed location information such as longitude, altitude, and heading; another three UI-related metrics are found in `ActivityManagerService` to log events such as process creation, process errors, and process termination. In total, we uncover 6 different types of metrics in `services.dex`, and 19 in `framework.dex`. In addition to general hooks in the framework, specialized hooks are also found in pre-loaded apps as well, particularly, the browser, the dialer, and (surprisingly) the calculator [4].

[3] In this version of firmware, the socket interface is disabled by default. But it can easily be enabled by restarting the service and passing it a parameter of `EnableSocket`. A recent firmware update from the vendor enables the socket interface by default.

[4] A hook is inserted into the `com.android.calculator2.CalculatorImageButton.onTouch` method to capture key press events. We suspect this hook is used just for testing.

Hook	Methods (Hooking Points)
callBrowserStopLoading	BrowserActivity.stopLoading
callOnPageStarted	Tab$2.onPageStarted
callOnProgressChanged	Tab$3.onProgressChanged
callOnReceivedError	Tab$2.onReceivedError
callOnReceivedTitle	Tab$3.onReceivedTitle
callReloadPage	BrowserActivity.onOptionsItemSelected BrowserActivity.resumeBrowser BrowserActivity$StopLoadingPageTimer.ResumeLoadingPage htc.ui.HtcTitleBar.onClick
callUserCancel	BrowserActivity.onKeyUp BrowserActivity.onOptionsItemSelected

Table 4: CarrierIQ hooks in the browser (to save space, we omit the common prefix of com.android.browser.htc.util. HTCBrowserIQAgent for the hooks, and com.android.browser for the methods.)

Table 4 lists the Carrier IQ hooks inserted into the browser to log many internal events. This information shows that the Carrier IQ software is intrusive and poses a serious threat to user privacy.

Case Study 3: Firmware Update During our experiments, the phone receives a firmware update that contains "important security updates". To locate what vulnerabilities are patched in this release, we upgrade the phone to the new release and compare it to the old one with DexDiff. The major change turns out to be the removal of the Carrier IQ software from the phone. Using DexDiff, we confirm that the agent and most of the hooks have been deleted from the phone. However, the hook in `Calculator.apk` has been missed and remains in the file. Also, vulnerabilities we discovered in `HtcDm.apk` have not been fixed while new functionality is added to it (hence more information can be leaked). The new firmware is based on a newer version of Android, which contains several security fixes. For example, the firmware adds the previously-missing capability to validate certificates against the PKI revocation list in `org.bouncycastle.jce.provider.PKIXCertPathValidatorSpi.engineValidate`.

3.3 Side-loaded App Verification

In a recent update to Android Jelly Bean (4.2.0), Google introduces the capability to scan side-loaded apps (i.e., apps installed from places other than the official app store) for malware. We use DexDiff to study how the side-loaded app verification is implemented. We find out that before side-loading an app, the app installer broadcasts an `android.intent.action.PACKAGE_NEEDS_VERIFICATION` intent to any apps providing the app verification service (e.g., the Google play app). To understand how the apps are verified by Google play, we use DexDiff to compare two versions of the app: version 3.8.17 from Jelly Bean 4.1.2, and version 3.9.16 from Jelly Bean 4.2.0. In the latter version of the app, a new Java package (`com.google.android.vending.verifier`) has been added. Instead of scanning the app locally, the verifier sends the identifying information of the app (e.g., package name, SHA256 hash, and origin, etc) to a remote server, and then receives a verdict from the server, similar to virustotal [54].

4. DISCUSSION

In this section, we discuss potential improvements to our system. First, DexDiff assumes that the official Android system is relatively safe and thus focuses on studying security of vendor customization. As demonstrated by previous and our own research, vendors tend to introduce subtle but serious vulnerabilities into the system through aggressive customization [13,19,29]. In this case, DexDiff provides a valuable comparative approach to audit the vendor customization. Nevertheless, the official Android could be vulnerable as well [40]. There have been a long stream of research that can be applied to improve security of Android [7, 10, 15, 17, 29, 31, 45, 59]. These systems and DexDiff have difference focuses and complement each other. For example, they can be used to scrutinize pre-loaded apps and libraries that do not have a counterpart in the official system. Moreover, although our experiment used a relatively old phone to demonstrate the effectiveness of our approach, the approach itself is generic and can be readily applied to newer phones and phones made by other manufacturers.

Second, like other similar systems, DexDiff makes various tradeoffs between accuracy and in the design that may affect its effectiveness in some specific cases. For example, DexDiff ignores registers in the bytecode. This mitigates the impact of register allocation of the Java and Dex compiler. However, some basic blocks can only be distinguished by registers. Ignoring registers makes these basic blocks indistinguishable. Similarly, DexDiff does not take edge labels into consideration when comparing two CFGs. This improves the system's tolerance to control flow type changes, but may be less accurate for some structures (e.g., the `switch` statement). Moreover, DexDiff utilizes Hungarian algorithm [57] to find an optimal assignment of classes and methods that maximizes the overall similarity. This optimizes global similarity but might not produce best-match for individual classes or methods. For example, assuming we have two binary files with classes A, B and $\mathcal{A}'$, $\mathcal{B}'$, the similarity between $A\mathcal{A}'$, $A\mathcal{B}'$, $B\mathcal{A}'$ and $B\mathcal{B}'$ are 100%, 80%, 80%, and 50%, respectively. With Hungarian algorithm, DexDiff will assign A to $\mathcal{B}'$, and B to $\mathcal{A}'$ (a total of 160%). However, the best match for A is instead $\mathcal{A}'$ (100%). In addition, DexDiff uses n-gram similarity to match classes and methods before comparing CFGs. This improves the scalability and efficiency of DexDiff by avoiding pairwise graph comparison for all the CFGs, but it might be less flexible if methods are "cut-and-pasted" across classes. Although we did not encounter such cases during our experiment, the issue can be mitigated by comparing a class to several most similar classes, instead of the only class returned by class matching.

Third, DexDiff compares two Android binary files at the bytecode level. We choose this approach because most of pre-loaded apps in a third-party phone are close-sourced due to their liberal license models (except for the GPL-licensed Linux kernel). Alternatively, we could first de-compile the apps/libraries and compare them against the base [16]. The same technologies in our paper (similarity and graph isomorphism) can be easily adapted to compare the de-compiled Java source code. On the other hand, the

usefulness of our system is not constrained by it because Dalvik bytecode contains adequate semantic information about the app.

Fourth, in its current form, DexDiff uses a simple method to locate the phone's base. Differences between the located and the actual base introduce noise into the system. It is desirable for the located base to be as close to the actual base as possible. We can use the fingerprinting technique to improve the accuracy in locating the base. Specifically, for each possible base (i.e., tags in the AOSP repository near the phone's reported version), we select a few unique changes as its fingerprint, and then try to locate (part of) the fingerprint in the phone. The phone's closest base is the most recent possible base whose fingerprint exists in the phone's software. Binary search can be used to accelerate the process.

Finally, DexDiff is a tool to pinpoint differences between two Android binary files. It can provide valuable inputs to (external) security analysts. However, it does not understand semantics of these differences. Much of the vulnerability identification still relies on human efforts and experiences. In the future, we plan to extend DexDiff with (some) automatic vulnerability detection capabilities to further reduce manual efforts.

5. RELATED WORK

Smartphone Security: the first area of related work is recent efforts in understanding and protecting security and privacy in mobile phones. Many systems have focused on detecting threats on the smartphone platforms. For example, TaintDroid [15] and PiOS [14] apply dynamic taint tracking [52] and static data-flow analysis to identify privacy leaks in Android and iOS apps, respectively. Stowaway [17] uses automated testing tools to map Android APIs to permissions, and then detects permission over-privilege for apps. PScout [7] analyzes the design and implementation of the Android permission systems. DexDiff focuses on analyzing security of customization made by the third-party manufacturers. ComDroid [10], Felt *et al.* [19], Woodpecker [29], and CHEX [45] employ static program analysis to detect confused-deputy [32] problems (or capability leaks) in Android apps. We also identified a similar problem in a pre-loaded system app with the help of DexDiff. Both Woodpecker [29] and DexDiff target pre-loaded apps and libraries, but have different focuses and approaches. DroidMOSS [59] and Juxtapp [31] are two closely related systems. They apply fuzzy-hashing and feature-hashing, respectively, based code similarity to detect repackaged Android apps. In addition to leveraging code similarity to match classes and methods, DexDiff further reveals structural differences between matched methods by comparing their control flow graphs. Our system thus provides more fine-grained differences of two apps, which benefits security audit more than a binary verdict of similar or not. Moreover, WHYPER [47] uses natural language processing to automatically assess the risk of mobile applications.

There are also various works focusing on malware detection of individual apps or the Android markets [16,18,23,30,60,61]. Enck *et al.* [16] decompile Android apps and applies Java-based program analysis to detect security issues in Android apps. DexDiff instead works directly on the Android binaries. However, our solution is not constrained by this because Dalvik bytecode contains rich type information. Moreover, the same techniques in DexDiff can be applied to compare Java source code. The MalGenome project [60] characterizes existing Android malware families to study their evolving trend. DroidRanger [61] applies behavioral pattern matching to detect malware in Android markets.

From another perspective, many works propose effective ways to enhance security of Android systems: QUIRE [12] and Felt *et al.* [19] allow apps to inspect the IPC call chain and, if necessary, drop privileges that the caller lacks to prevent the confused-deputy attack; AdDroid [48], AdSplit [53], APEX [46], AppFence [33], and TISSA [62] extend the Android system to fine-tune app's access to private information. For example, AdDroid [48] and AdSplit [53] separate the in-app advertisement libraries from the host app so privileges can be granted differently for them. Cells [4] and L4Android [44] build efficient virtualization platforms for Android, allowing multiple virtual phones to securely run on a single physical device, isolated from each other. While all these systems enhance the Android system security, DexDiff has a different goal of detecting unsafe manufacturer customization. They can be naturally combined to provide defense-in-depth.

Code Similarity and Binary Diff: the second area of related work includes systems to detect code similarity and compare binary files. Code similarity has been employed in various security applications such as malware classification [35, 38], code plagiarism detection [8,51], (unpatched) code clone detection [37,39,41], and Android repackaged app detection [31, 59]. For example, BitShred [38] leverages feature hashing based code similarity analysis to enable large-scale malware triage and clustering. ReDeBug [37] uses feature hashing to quickly detect unpatched code clones in large code bases. MOSS proposes a technique called winnowing based on fuzzy hashing to generate fingerprints for documents (including source code). MOSS is a popular tool to detect plagiarism in programming assignments. In comparison, DexDiff additionally provides structural differences for two Android binaries as required by our goal.

Binary diff has been used widely to analyze malware, vulnerabilities, and improve (or exploit) patches. For example, BinDiff [20] compares two binaries by using heuristics to identity a common isomorphic subgraph. In comparison, DexDiff extends the backtracking algorithm to construct more reliable maximum common subgraphs. BinHunt [21] is a closely related system. It uniquely leverages symbolic execution to semantically compare basic blocks and then uses the backtracking framework to compare two `x86` binaries. DexDiff instead targets the Android binaries. It applies code similarity to first coarsely classify classes and methods, saving the expensive graph isomorphism only for CFG comparison. We adapt backtracking for Android binaries as well to make the exhaustive algorithm practical for even large Android binaries. Given a vulnerable program and its patched version, Brumley *et al.* [9] propose to automatically generate exploits that target essential differences between these two versions of the program. There are also binary update tools (e.g., bsdiff & bspatch [49] and Courgette [28]) that generate and apply patches in binary differences. Patches produced by these tools have substantially smaller size, thus improving efficiency and timeliness of patch distribution. Courgette (also known as Google update) is more efficient than bsdiff by first transforming the programs into an intermediate format in which binary diffing is more effective. However, these tools do not provide capabilities required by Android binary auditing.

6. SUMMARY

Android has become the leading mobile platform due to the wide availability of third-party Android phones. To differentiate their products, vendors often deeply customize their phones, leading to vulnerabilities that do not exist in the official Android system. In this paper, we propose to systematically audit vendor customization by comparing the phone side-by-side to its base. To facilitate these efforts, we designed DexDiff, a system that can pinpoint fine structural differences of two Android binaries and further present them in the surrounding context. DexDiff allows external security analysts (without source code access) to focus on modifications

that are more likely to introduce vulnerabilities. It first coarsely matches classes and methods in the input Android binaries with *n*-gram based code similarity, and then generates fine-grained structural comparisons by identifying the maximum common isomorphic subgraphs of the CFGs for each pair of matched methods. We have built a prototype of DexDiff and applied it to a popular commodity Android phone. Our evaluation demonstrates that DexDiff is efficient, scalable, and effective in identifying unsafe vendor customization.

7. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insightful comments that helped to improve the presentation of this paper. We also want to thank Yajin Zhou, Gary Tyson, and Xuxian Jiang for the helpful discussion. This work was supported in part by the First Year Assistant Professor award of Florida State University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of FSU.

8. REFERENCES

[1] An Assembler/Disassembler for Android's dex Format. http://code.google.com/p/smali/.

[2] Subgraph Isomorphism Problem. http://en.wikipedia.org/wiki/Subgraph_isomorphism_problem.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Prentice Hall, 2006.

[4] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: a Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd SOSP*, 2011.

[5] Apache. Apache Harmony: Open Source Java Platform. http://harmony.apache.org/.

[6] AT&T. Graphviz - Graph Visualization Software. http://www.graphviz.org/.

[7] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 19th ACM CCS*, 2012.

[8] Brenda S. Baker. Deducing Similarities in Java Sources from Bytecodes. In *Proceedings of the 1998 USENIX ATC*, 1998.

[9] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 29th IEEE S&P*, 2008.

[10] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th ACM MobiSys*, 2011.

[11] Anthony Desnos. androguard:Reverse engineering, Malware and goodware analysis of Android applications ... and more (ninja !). https://code.google.com/p/androguard/.

[12] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[13] Trevor Eckhart. CarrierIQ. http://androidsecuritytest.com/features/logs-and-services/loggers/carrieriq/.

[14] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the 18th NDSS*, 2011.

[15] William Enck, Peter Gilbert, 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. In *Proceedings of 9th USENIX OSDI*, 2010.

[16] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[17] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM CCS*, 2011.

[18] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A Survey of Mobile Malware in the Wild. In *Proceedings of the 1st ACM SPSM*, 2011.

[19] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[20] Halvar Flake. Structural Comparison of Executable Objects. In *Proceedings of the 1st DIMVA*, 2004.

[21] Debin Gao, Michael K. Reiter, and Dawn Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Proceedings of the 10th ICICS*, 2008.

[22] Gartner. Gartner Says Worldwide Sales of Mobile Phones Declined 2.3 Percent in Second Quarter of 2012. http://www.gartner.com/it/page.jsp?id=2120015.

[23] Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. Vision: Automated Security Validation of Mobile Apps at App Markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, 2011.

[24] Google. Android Developers. http://developer.android.com.

[25] Google. Android Device Gallery. http://www.android.com/devices/?f=phone.

[26] Google. Android Open Source Project. http://source.android.com.

[27] Google. Dalvik Technical Information. http://source.android.com/tech/dalvik/.

[28] Google. Software Updates: Courgette. http://dev.chromium.org/developers/design-documents/software-updates-courgette.

[29] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th NDSS*, 2012.

[30] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th ACM MobiSys*, 2012.

[31] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *Proceedings of the 9th DIMVA*, 2012.

[32] Norman Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22, October 1998.

[33] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These Aren't the Droids You're Looking For: Retroffiting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM CCS*, 2011.

[34] HTC. HTC EVO 4G. http://www.htc.com/us/smartphones/htc-evo-4g-sprint/.

[35] Xin Hu, Tzi cker Chiueh, and Kang G. Shin. Large-Scale Malware Indexing Using Function-Call Graphs. In *Proceedings of the 16th ACM CCS*, 2009.

[36] Intel. Intel 64 and IA-32 Architectures Software Developer Manuals. August 2012.

[37] Jiyong Jang, Abeer Agrawal, and David Brumley. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *Proceedings of the 33rd IEEE S&P*, 2012.

[38] Jiyong Jang, David Brumley, and Shobha Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proceedings of the 18th ACM CCS*, 2011.

[39] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th ICSE*, 2007.

[40] Xuxian Jiang. SEND_SMS Capability Leak in Android Open Source Project (AOSP), Affecting Gingerbread, Ice Cream Sandwich, and Jelly Bean. http://www.cs.ncsu.edu/faculty/jiang/send_sms_leak.html.

[41] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 2002.

[42] Evgeny B. Krissinel and Kim Henrick. Common Subgraph Isomorphism Detection by Backtracking Search. *Software–Practice & Experience*, 2004.

[43] Eric Lafortune. ProGuard. http://proguard.sourceforge.net/.

[44] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4Android: A Generic Operating System Framework for Secure Smartphones. In *Proceedings of the 1st ACM SPSM*, 2011.

[45] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 19th ACM CCS*, 2012.

[46] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In *Proceedings of the 5th ACM ASIACCS*, 2010.

[47] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the 22th USENIX Security Symposium*, 2013.

[48] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the 7th ACM ASIACCS*, 2012.

[49] Colin Percival. Naive Differences of Executable Code. http://www.daemonology.net/bsdiff/.

[50] Artem Russakovskii. Massive Security Vulnerability In HTC Android Devices (EVO 3D, 4G, Thunderbolt, Others) Exposes Phone Numbers, GPS, SMS, Emails Addresses, Much More. http://www.androidpolice.com/2011/10/01/massive-security-vulnerability-in-htc-android-devices-evo-3d-4g-thunderbolt-others-exposes-phone-numbers-gps-sms-emails-addresses-much-more.

[51] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD*, 2003.

[52] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *Proceedings of the 31rd IEEE S&P*, 2010.

[53] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating smartphone advertising from applications. In *Proceedings of the 21th USENIX Security Symposium*, 2012.

[54] VirusTotal. VirusTotal - Free Online Virus, Malware and URL Scanner. http://www.virustotal.com/.

[55] Wikipedia. Assignment Problem. http://en.wikipedia.org/wiki/Assignment_problem.

[56] Wikipedia. Feature Hashing. http://en.wikipedia.org/wiki/Feature_hashing.

[57] Wikipedia. Hungarian algorithm. http://en.wikipedia.org/wiki/Hungarian_algorithm.

[58] Allan Wojciechowski. DexDiff. https://github.com/allanwoj/DexDiff.

[59] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the 2nd ACM CODASPY*, 2012.

[60] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE S&P*, 2012.

[61] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th NDSS*, 2012.

[62] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, 2011.