Divide and Conquer: Recovering Contextual Information of Behaviors in Android Apps around Limited-quantity Audit Logs

Zhaoyi Meng*, Yan Xiong†, Wenchao Huang*, Fuyou Miao*, Taeho Jung† and Jianmeng Huang*

* School of Computer Science and Technology, University of Science and Technology of China
  mzy516@mail.ustc.edu.cn, {yxiong, huangwc, mfy}@ustc.edu.cn, mengh@mail.ustc.edu.cn
† Department of Computer Science and Engineering, University of Notre Dame, tjung@nd.edu

Abstract—We propose and implement DroidHolmes, a novel system that recovers contextual information of app behaviors around limited-quantity audit logs. The key module of DroidHolmes is identifying the path matched with logs on the app’s control-flow graph (CFG). The challenge, however, is that the limited-quantity logs may incur high computational complexity in the log matching, where there are a large number of candidates caused by the coupling relation in matching successive logs. To address the challenge, we propose a divide and conquer algorithm to individually position each node on the CFG matched with logs. In our experiments, DroidHolmes recovers contextual information in the behaviors of real-world apps. Meanwhile, DroidHolmes incurs negligible performance overhead on smartphones.

I. INTRODUCTION

Android users are now suffering serious threats from unwanted behaviors of various apps. The analysis of audit logs is one of the most important methods for device manufacturers to unveil the underlying malware of app behaviors.

Many techniques [1], [2] have been proposed to assist analysts in reconstructing app behaviors based on runtime logs. However, there still exists a problem that is the lack of contextual information within the reconstructed behaviors. Specifically, it is impractical to deploy runtime mechanisms with heavyweight computations for behavior reconstruction due to the resource-constrained feature of the Android smartphone. Instead, analysts choose to log some critical points for reconstructing coarse-grained behaviors, while the contextual information (e.g., guarding conditions of sensitive actions, information-flow transmission paths, API usage) is unavailable. Actually, the contextual information is valuable evidence to unveil the intentions of app behaviors.

We propose and implement DroidHolmes, a novel system that recovers contextual information of app behaviors around audit logs [3]. DroidHolmes gathers runtime logs about an app from the Android middleware and then identifies the path matched with the logs from the app’s CFG on the PC. The path is used to extract required contextual information by existing analysis methods. DroidHolmes only requires limited-quantity logs to recover the contextual information, and meanwhile, the overhead imposed on mobile devices is negligible. Our goal is reconstructing context-aware app behaviors to help analysts identifying the malice of apps.

The major challenge of implementing DroidHolmes is the high computational complexity of the log matching. It is caused by the coupling relation that when a node is matched, its successors are the candidates for matching subsequent logs. When some mechanisms (e.g., reflection and obfuscation) are abused to evade malware detection [4], the number of possible candidates increases exponentially. A straightforward method is recording enough logs to distinguish each branch, but the runtime overhead imposed on the smartphone is considerable. Existing static analysis tools [5] cannot parse obfuscated or encrypted arguments so far.

We propose a divide and conquer algorithm to address the challenge. Our key idea is leveraging the call stack information at runtime to decompose the coupling relation in the log matching. The information can be used to individually locate each node matched with each log record on the CFG. In this way, the computational complexity is reduced.

II. DROIDHOLMES’S DESIGN

A. Architecture

Figure 1 depicts the overall architecture of DroidHolmes:

Logging Module (on Android Platform)
- Using Apps
- Application Level
- Logging API
- Android Middleware

Auditing Module (on PC)
- Path Generation
- Graph Building
- Log Matching
- Path Identifying
- Context Extraction
- Data-flow Analysis
- API Misuse
- Trigger Analysis
- etc.

The logging module is used to record audit logs when a user interacts with apps on the smartphone. The module is deployed in the Android middleware to capture the invocations of specified Android APIs. The logs are sent to the auditing module with the detected app.

Auditing Module. The auditing module is run on a PC to recover contextual information around the logs. It analyzes the app’s code to build the CFG at first. Upon the graph, it identifies the path matched with the logs. Finally, analysts can leverage existing techniques (e.g., data-flow analysis, trigger analysis, manual inspection) to extract required contextual information from the path.
B. Log Matching

1) Logging Information: The core element of a log record is a tuple \((\text{Des}, \text{Csi})\), where \text{Des} is the description of the invoked API and \text{Csi} is the call stack information when the log record is produced.

\text{Des} contains the signature of the logged API and the used arguments. The signature is used to distinguish different app operations, and the arguments are used to decide the successors when some APIs are related to some Android mechanisms, e.g., reflection and ICC.

\text{Csi} is the key information to position matched nodes. It includes the last \(K\) methods pushed in the stack \((P)\) and the depth of the stack \((D)\). With the elements, the search space of matching each log record is confined in a method.

2) Matching Strategy: We propose a divide and conquer algorithm to identify the path matched with logs. The algorithm is designed upon the depth-first search and uses the call stack information to guide the path exploration.

Divide. We use \(D\) to confine the search depth for matching the target log record and use \(P\) to indicate the method sequence for finding the method including the logged API callsite. In other words, the algorithm stops searching forward and performs a backtrace to traverse another branch in the following two cases: 1) the current search depth exceeds \(D\); 2) the method to be visited is mismatched with the required method in \(P\).

Conquer. When the search space of matching the target log record is confined in a definite method, we perform the depth-first search on the method’s CFG to match the signature of each invoked API with the signature in \text{Des}. If the matched API callsite is reflection- or ICC-related, the arguments in \text{Csi} can be used to decide successors for matching the following logs.

III. EXPERIMENTAL RESULTS

We select a representative app to evaluate DroidHolmes. It is a malware sample of FakeInstaller family, whose MD5 is dd40531493f53456c3b22ed0bf3e20ef. In the app’s code, almost all the methods are invoked by reflective calls and the arguments of the calls are obfuscated. It is challenging to reveal the intention of the app with existing tools [5].

DroidHolmes introduces negligible runtime overhead on the Android smartphone (2.39% on average). It spends 640.1 seconds to identify the path matched with 17097 log records and the memory consumption is 26.6 GB. As a comparison, we use the depth-first search without our algorithm to find the matched path. The result of the downgraded strategy is that no matched path is found within 5 hours.

We manually analyze the identified path to extract detailed contextual information of the app behavior. Figure 2 depicts a path segment with recovered contextual information, where the path on the left is built by sequentially connecting the logs, and the path on the right is the result of DroidHolmes. Specifically, we notice that the library embedded in the app collects user’s private information (i.e., IMEI, IMSI) by calling \text{getSubscriberId()} and \text{getDeviceId()} when the app launches. The app then checks if each character of the collected strings can be transferred to a value with the integer type. It may be an anti-virtualization technique to avoid obtaining mock information. When the checks pass, the app transmits the information to the specified server by the network. There is another suspicious behavior that the app invokes \text{toCharArray()} after obtaining the method’s signature by recombining many unordered characters, which is a trick to evade static analysis.

ACKNOWLEDGEMENT

The authors would like to thank all anonymous reviewers for their constructive advice. The research is supported by National Natural Science Foundation of China under Grant No.61572453, No.61202404, No.61520106007, No.61170233, No.61232018, No.61572454. Wenchao Huang is the corresponding author.

REFERENCES

[1] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, “Copperdroid: Automatic reconstruction of android malware behaviors,” in NDSS 2015.

[2] X. Yuan, O. Setayeshfar, H. Yan, P. Panage, X. Wei, and K. H. Lee, “Droidforensics: Accurate reconstruction of android attacks via multi-layer forensic logging,” in ASIACCS 2017. ACM, 2017, pp. 666–677.

[3] Z. Meng, Y. Xiong, W. Huang, F. Miao, T. Jung, and J. Huang, “Divide and conquer: Recovering contextual information of behaviors in android apps around limited-quantity audit logs,” arXiv preprint arXiv:1809.07036, 2018.

[4] S. Rastrohofer, S. Arzt, M. Miltenberger, and E. Bodden, “Harvesting runtime values in android applications that feature anti-analysis techniques.” in NDSS, 2016.

[5] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, “Droidra: Tampering reflection to support whole-program analysis of android apps,” in Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, 2016, pp. 318–329.