Developing a Zen Click Fraud Detection Framework that Uses Smart Contracts

Sean Sanders
University at Buffalo
spsander@buffalo.edu

Lukasz Ziarek
University at Buffalo
lziarek@buffalo.edu

Abstract

Over 3,739 apps on average are published per day on the Google Play store [1]. A handful of the applications contain advertisement malware referred to as malvertising. As a result, Android advertisement malware has been a growing multi-billion-dollar problem. It constantly assaults many of the major advertising libraries such as the Google, Facebook, and Amazon. This paper presents an effective strategy for countering advertising malware using dynamic and static analysis techniques and the Soot compiler framework. Our research aims to detect malvertising click fraud in Android applications using the Soot compiler framework and blockchain technology. But the approach and the framework can be used to counter mobile malware families.

1. Introduction/Motivation

Over the past several years, Google Play has released around 100,000 Android apps per month [2]. Mobile malware has been a significant global problem. According to McAfee, the development of mobile malware is on the rise [3]. Based on their quarterly report in 2019, they found that a total of 35 million mobile malware programs have been produced. In essence advertising click fraud is a prevalent and growing threat to mobile advertisers.

Android advertising malware (malvertising) involves fraudulent behavior related to advertising libraries. There are two primary strategies employed by advertising malware. The first involves the displaying of fake advertising or hiding ads. The other technique involves generating phony ad impressions. Malvertising also includes the clicking of and ad without user interaction. This typically occurs where a malicious entity inserts malware code into an Android application. There are numerous related malware families and malware versions that utilize the primary strategies. The top ten malware variants equate to 77% of malware activity in January 2021 [4].

App publishers need tools to counter the identified and not-yet-identified malicious developers who are coding applications that will engage in click fraud, collect private information, and establish ransomware beachheads. The toolsets being developed and demonstrated are cutting-edge automated tools (refer to the terminology table in the Appendix Terminology for all definitions of terms used).

There are numerous domains where these tools and technology can be applied. Specialized malware forensic analysis tools could be developed for

1. Publishers needing a massive screening tool for a quick review of any potential threats.
2. Publishers who are suspicious of the Android code and want to insert auditing code to track app state behavior.
3. Businesses and educational institutions where users request app installation.

This research focuses on how to build and develop an auditing framework to detect malvertising click fraud found in the Zen malware family. To the best of our knowledge, using multiple components such as online malware identification, static and dynamic analysis, and blockchain is a unique approach for detecting click fraud. This is a unique approach because it requires the integrations of several complex components including the Soot compiler framework, blockchain smart contracts, and the understanding intricate auditing infrastructures.

1.1. Research Problem

The Zen Malvertising family is a complex system of malware that has been known to participate in advertisement click fraud in Android applications. The Zen malware is part of the PHA (potentially harmful application) family and is a relatively new family of malware.

URI: https://hdl.handle.net/10125/80230
978-0-9981331-5-7

The Zen malware is dangerous because it uses root privileges to gain access for inserting a rooting trojan utilizing command-and-control servers. Command-and-control servers are engine used by the Zen family to communicate with the malware that resides in the infected Android applications. The command-and-control servers are typically used to tell the application what actions to perform and when to install malicious files on the Android device.

The Zen malvertising family's complex nature and behavior have led to the development of the forensic auditing framework described in this paper. The framework consists of an online component that assists in quickly identifying the Zen malware associated with the specific Android applications. The static component involves detecting the location in the app to inject the blockchain calls. The dynamic analysis component helps identify if any click fraud occurred in the Android application that has been identified as containing Zen malware.

1.2. Research Question

Based on the prevalent issue of malvertising click fraud, this leads to the following question.

The primary research question is: *Is it possible to model the Zen malware advertising family and create a blockchain smart contract encoding that will store the data and help forensic analysts with the detection of the Zen malware family clicks fraud?*

2. Related Work

To our knowledge, using the blockchain as a mechanism for storing the advertisement fraud information and helping with the detection of advertising fraud behavior is a new and unique approach. While it is unique, there are several other detection systems that have been developed.

For example, there are four different strategies for classifying malware using machine learning approaches [5, 6]. The techniques include using static or dynamic analysis, using both static and dynamic analysis, using string analysis, or using classification techniques. However, most of the machine learning classification approaches do not deal with mobile advertising fraud [7–15]. While most of the papers focus on classifying malware, they do not focus on mobile ad fraud detection.

Several papers discuss the role of machine learning to discover and classify malware in Android applications [5, 6, 16, 17]. Fung et al. created a tool called RevMatch. The framework uses machine learning to assist with the detection of malware [16]. RevMatch is a collaborative malware decision tool that queries the labeled malware detection history from a database. If limited information is returned from the query, then the system uses partial matches to make decisions on what the malware is classified as. The authors compared multiple machine learning techniques to compare the false positives and errors that result from each technique.

The smart contract approach described here could benefit from RevMatch's approach for malware detection by incorporating the RevMatch queries approach into the Soot analysis framework. Their approach would allow us to have a mechanism in place to assist with deciding if a click was legitimate or not legitimate during the dynamic analysis phase.

Hurrier et al. discuss the Euphony malware classification tool [17]. Euphony uses malware labels to classify malware, these labels are then applied to each application using machine learning classification taxonomy. The Euphony tool essentially assigns the Android malware samples to malware families. One of the benefits of this approach is that it does not require prior knowledge of malware families for the classification process. This approach is particularly useful for our implementation because the Euphony method can be directly applied to detect click fraud.

Shijo et al. introduce a method that uses static and dynamic analysis techniques to detect malware in Windows applications [6]. Their static analysis method requires the extraction of printable string information (PSI) and these strings can be used to assist with detecting a malicious command-and-control servers. Another nice feature of the Shijo et al. approach is that they use machine learning vectors to assist with the detection of malware using test and training datasets [18]. Vectors in machine learning are tuples of one or more scalar values. They also reported interesting empirical results.

Their experimental results show an accuracy of 95.8% using static analysis, 97.1% using dynamic analysis, and 98.7% using the integrated method to predict malware threats. The Shijo et al. paper provide an interesting solution for the application of using string information and strings to help detect malware. String analysis involves analyzing all of the stings inside an Android application for potential threats. It would be challenging to use this approach in our auditing framework. However, it might be possible to adapt this approach to the Soot framework by modifying the process rules for backwards and forwards flow analysis.

Vecchio et al. developed a machine learning solution for classifying malware using graph structures of the strings that were created [5]. Their system uses a three-step process for detecting malware. The first step involves using static analysis to extract the strings. In the next step, they use a feature space generator to extract the compiler computations. In the final step they use k-fold cross-validation and multiple machine learning algorithms to assist with the malware classification.

The novel approach by Vecchio et al. achieves a recall rate of 97% for classifying when an application contains malware. It is theoretically possible that our approach could adapt their approach and use string analysis and string extraction to detect click fraud. In summary leveraging the various machine learning algorithms to assist with the detection of click fraud in our Soot based smart contract framework has potential. In particular they could assist with identifying command-and-control servers.

Data flow analysis for tracking malware is another common technique employed in the wild west of malware detection. Fuchs et al. created an analysis tool that facilitates automatic reasoning about the security of Android applications [5]. The approach performs incremental checking of the application and extracts the Android applications' manifest file, checks the security specifications, and ensures compliance based on the data flows. It is possible that our implementation could use this approach to assist with detecting advertisement click fraud through analyzing the data flows and where to exactly inject the code. Data flow analysis is particularly useful in detecting command-and-control servers.

Beaucamps et al. developed an alternative approach to detecting malware via the abstraction of application behaviors [19]. The behaviors were abstracted by dynamically examining the program traces. Suspicious behaviors were detected by comparing trace abstractions to reference malicious behaviors. The authors opted to have the execution traces represented as a trace automation. The traces were reconstructed, which then produces a representation, which is independent of the program flow. Our framework could use this in our auditing framework to assist with identifying dynamic abstraction of program traces and behavior.

3. Zen Malware Family Click Fraud

The Zen malware family requires root privileges on Android devices to work correctly [20]. The root privileges allow the application to turn on accessibility service (a service used to allow Android users with disabilities to use their devices) for itself. This is accomplished via writing to the Android security mechanism called *enabled_accessibility_services*.

Zen, however, doesn't check for root privileges; it assumes it has it. Zen implements three accessibility services directed at different Android API levels and uses these accessibility services, chosen by checking the operating system version, to create new Google accounts [20]. This is accomplished via opening the Google account creation process and parsing the current view. The app will click on the appropriate buttons and input boxes and sign up and it does not require user interaction.

The developers of the Android security system implemented a CAPTCHA process to stop hackers from creating new accounts. Bypassing the security mechanism requires that the app use its root privilege to inject code into the Setup Wizard, then extract the CAPTCHA image, and then send it to a remote server to solve the CAPTCHA. Note, the Zen Trojan does not implement any kind of code obfuscation except for the use of one string that is linked to a server. The code obfuscation string uses a Base64 encoding.

The malicious Android app also injects its own code into the *system_server* process, which requires root privileges [20]. This is most likely done to hide from any anti-PHA systems that look for a specific app process name. Or to hide from the Android app when it examines memory to identify malicious processes. The malware typically creates hooks to prevent the phone from going to sleep, booting, or restarting. These hooks are created by the Zen malware using the root access and a custom native code called *Lmt_INJECT*. The process used by the Zen malware family to infect a system is demonstated in Figure 1. The first step used by the Zen family is to turn off SELinux protection. The SELinux protection is as a security mechanism to combat malicious applications in Android applications. In the next step, the app identifies an Android process ID value to inject with code (this is accomplished via a series of *syscalls*). The ”source process” refers to the Zen trojan running as root, while the ”target process” refers to the process to which the code is injected and refers to the target process *pid* value [20].

4. Creating an Auditing Framework (Proposed Solution/Methodology)

Creating an auditing framework consists of integrating multiple components including the Virus Total, static and dynamic analysis (Figure 2). The first phase referred to as the online component related to virus identification. This phase requires utilizing VirusTotal to help classify the Android application's malware family. Classifying malware requires the use of the original Android application's hash value. The

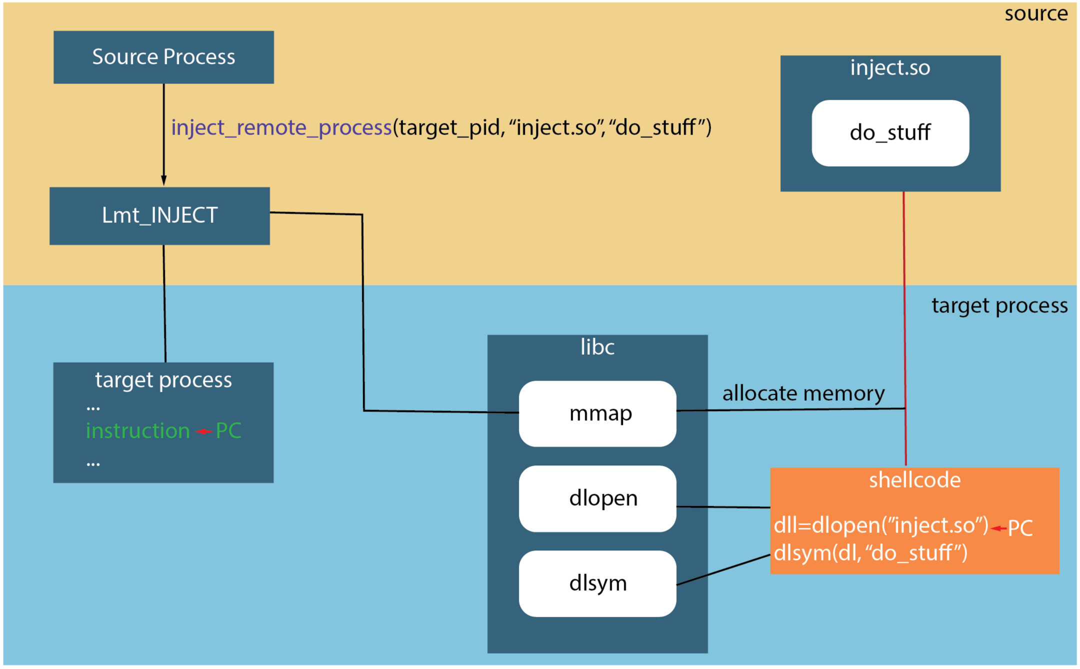


Figure 1. Zen overview Adapted from [20]

hash will then be sent to VirusTotal. VirusTotal will then reveal the malicious details associated with an Android application.

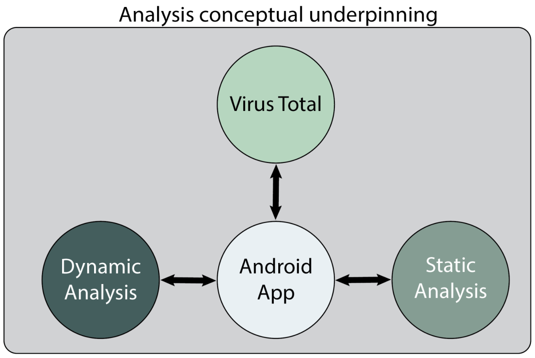


Figure 2. Conceptual underpinning of the analysis

The second phase is the static analysis, where the Android application has to be analyzed and where it is necessary to determine where to inject the blockchain calls. This requires that the Soot framework detects where the ad clicks occur and then injects blockchain calls. This step requires that there is a scan phase that looks for the possible command-and-control servers and the code associated with clicks. We discovered through manual analysis that both Google and Amazon Ad's library has the same performClick function call. The function call is associated with the clicking of an advertisement. This makes it much easier to identify clicking behavior. The third phase of the forensic framework is the dynamic analysis. In this phase the Android application is actually running and sending the legitimate and illegitimate click fraud information to the Ethereum smart contract on the blockchain. The last step of the process monitors users interactions with the auditing system.

All the coding details for the entire process can be found at: (`https://github.com/SeanSandersPersonal/Zen-Click-Fraud-Detection-Framework`).

4.1. Android Application Code Injection

There are two malvertizing fraud approaches used by the Zen malware family. The first malicious interaction involves a command-and-control server attacks. In this approach the Zen server sends commands or program code to the application to instruct the application to engage in advertisement click fraud. The second malicious interaction by the Zen family involves using a custom advertising library to route the traffic from other domain locations. This in effect tricks the advertisers into thinking that the application advertisement came from one legitimate application, where in reality it came from many locations.

4.2. Decompiling Android applications and Code Analysis

The Soot framework is the foundational platform for analyzing the Zen malware family. The Soot framework is to identify where the code clicks are occurring is a powerful technique for preventing or deterring advertisement click fraud. The Soot framework uses an intermediate representation of the original Android Java code (Jimple) and allows for the re-compiling of the Android application with minimal effort. However, there is a significant problem. The hard task for our approach is determining where to inject the Ethereum blockchain smart contract calls. Determining where to inject the blockchain calls can be remedied by focusing on how the Zen malware family behaves. Below is a list of common Zen malware behaviors.
Zen malware behaviors include:

1. Programming the clicking of advertisements without user interaction

2. Using a command-and-control server to execute or perform clicks without user interaction

The first problem of detecting clicks without any user interaction is difficult because it requires understanding how the click fraud was coded. Understanding how clicks are performed is important because it will assist in understanding app behavior.

When using the Amazon Ad library to click on an advertisement, the user initiates or performs the click function. In the case of the Zen malware, this is accomplished using the performClick function. Then to ensure the ad is clicked, the *setPressed* function must be set to true. Both of the functions mentioned must be placed in a **try catch** statement in order to work.

When using the Soot framework, the forensic analysis tool should pinpoint the location of the *performClick* and *setPressed* function calls. When performing the analysis, it is necessary that the function calls only interact with the Amazon adView and not another library. Note also, that *performClick* and *setPressed* can be performed on buttons that are present in any Android application. Once both *performClick* and *setPressed* functions are found, the blockchain call should be placed after the *setPressed* function call. This is a critical step in the smart contract injection process.

Now let us focus on the second behavior, using a command-and-control server to execute or perform clicks without user interaction. Identifying code in Android applications command-and-control server that performs clicks can be very tricky. The primary reason is that the bad guys can hide the communication and use several linked libraries. For example, a linked library could involve a third party library that makes calls to Amazon's ad library. The Soot framework is very adept because the framework can perform many types of static and dynamic analysis to detect these linkages.

The Soot framework has strong techniques for conducting forward-flow analysis. Forward flow analysis provides information about the future code and paths of execution [21]. In essence, this means that it is possible to check all the execution paths that exist. In effect, this would provide insight into the paths leading from and to the advertising libraries.

The next step in the Soot framework process involves checking for all strings that contain external IP addresses. This is a very labor intensive task using the Soot framework. It was discovered that using the string analysis approach has lead to insights into how to easily discover the external IP addresses processes quickly.

The final step using the Soot framework involves checking all of the saved execution paths and the external IP addresses that Soot found. All of the external IP addresses have to be sifted through to find out exactly whether a command-and-control server was used inside of the Zen malware. Finally, the VirusTotal website `https://www.virustotal.com/gui/` can be used to help with sifting through malicious IP addresses that have been identified as being malicious.

4.3. Smart Contract Programming

In our approach, Ethereum Remix was used to create the smart contract. Remix is a powerful tool that allows users to program their smart contract through an online graphical user interface and it enables them to push their smart contracts to the private blockchain [21]. Remix also allows users to test their smart contracts in real time. This is very useful for individuals that are new to smart contract programming.

4.4. Retrieving Data from The Ethereum Blockchain

The rest of the discussion in this section will focus on the smart contract code for retrieving information from the smart contract struct in the blockchain. The following code allows the users to retrieve the information from the variables in the struct.

4.5. Blockchain Injection

Finding the appropriate place to inject smart contract code into the Android application is the most difficult task of monitoring and developing smart contract code for the forensic tool. The question that must be answered is where to place the appropriate blockchain calls in the Android application. To answer this question, the jimple code must be extensively examined and mapped.

Specifically, the process involves looking for Jimple labels that consistently interact with the Android framework and have the potential to be threats. Jimple uses labels for the *try* and *catch* statements. The objective is to look for the keyword performClick function and bookmark the statement as having a possibly malicious intent (Figure 3).

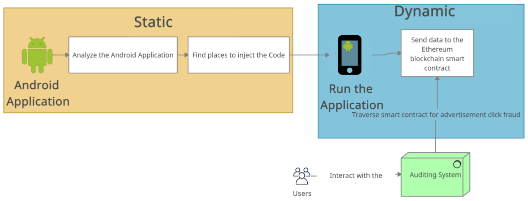

Static
Android Application
Analyze the Android Application
Find places to inject the Code
Dynamic
Run the Application
Send data to the Ethereum blockchain smart contract
Traverse smart contract for advertisement click fraud
Users
Interact with the
Auditing System

Figure 3. Flow overview

The next step involves identifying the *setPressed* function. Knowing the names of the advertisement clicking actions is helpful because the soot framework provides the user with the ability to get function names. The locations where the *performClick* and *setPressed* functions reside can then be logged internally. On the second pass of the Soot framework it is then possible to inject the *IncrementNonUserClickCount* blockchain function call into the Jimple code.

Initialization of the smart contract struct is necessary. This is accomplished by injecting the smart contract function call that initializes the struct inside of the MainActivity. Inside the function parameter we have to pass the Android application name as a parameter. The SetAppName blockchain function call requires that the Android application name is passed into the function.
In our example we use the following Java code: Resources appR = ctx.getResources(); CharSequence txt = appR.getText(appR.getIdentifier (”app$_{n}ame”,”string”,ctx.getPackageName()));$

The final step of the injection process invovles retrieving the app name from the Android application. This requires that the txt variable to be passed to the blockchain function call. This code would then be inserted at the beginning of the *OnCreate* method in the *MainActivity* of the Android application. To successfully find the *MainActivity* class, the Soot framework will have to find the entry points for the Android application. This requires that the dynamic entry points are extracted from the Android application by getting the reachable methods. The reachable methods are those which Soot can retrieve and have an active body. An active body in Soot is the method body that Soot retrieves from the Android application.

4.6. Malware Injection Leads to the Auditing Process

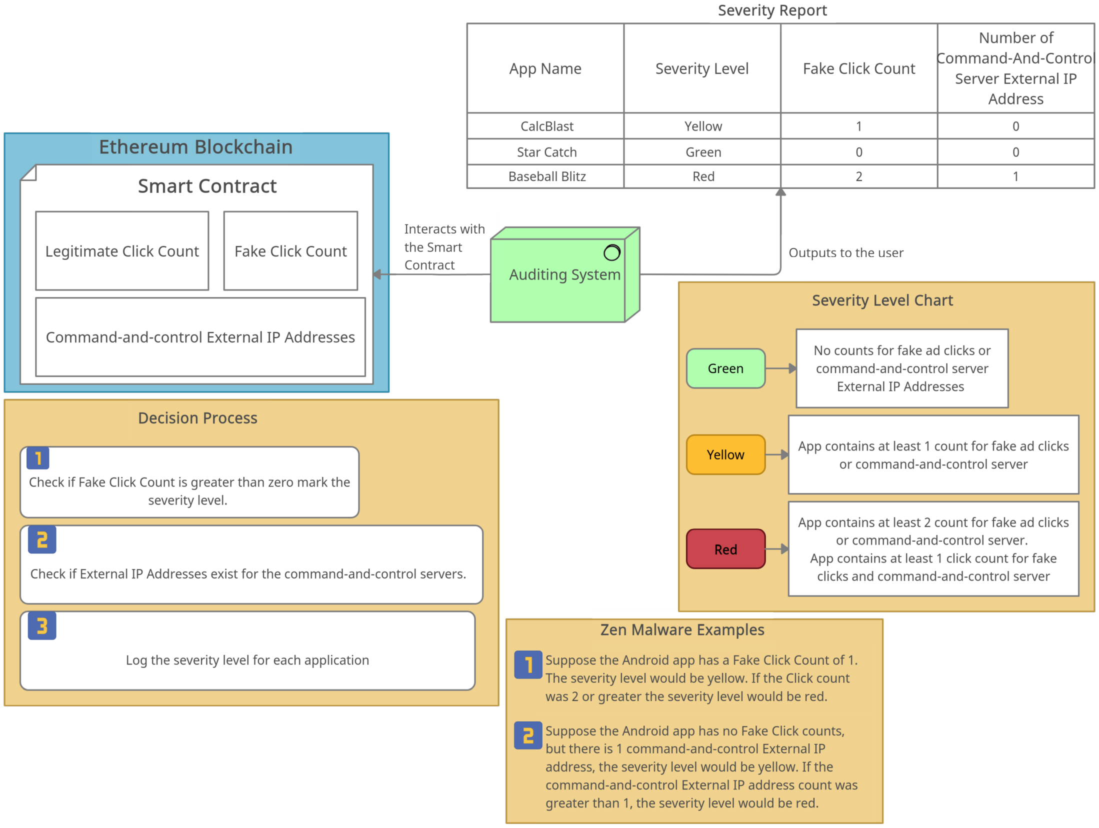
The first step in the auditing decision process involves the injection of smart contract blockchain calls into applications using Soot (Figure 4). This is necessary to ensure that the data is added to the smart contract in the Ethereum blockchain. The next step involves the interaction of the auditing system with the Ethereum blockchain.

This enables the construction of a multi-layered auditing reporting system. The auditing system is constantly communicating with the smart contract. This ensures that security specialists have the most up-to-date information when examining the severity report that is produced by the auditing system. The decision process involves analyzing the fake advertisement click counts and the number of external command-and-control IP server addresses involved. This information is then used to log the severity level. There are three severity levels, green, yellow, and red used by the forensic analysis framework.

The green level occurs when there are zero fake ad clicks and zero for the number of command-and-control server external IP addresses. A yellow level is logged for an application when it has at least one fake ad click count or one command-and-control server external IP address count. The yellow alert indicates caution. A red level alert is logged when there are more than one fake ad count or more than one command-and-control server with an external IP address. A red alert can also be indicated when there is a combination of one fake ad click and one command-and-control server with an external IP address.

In our forensic analysis framework, the data can be retrieved using Java or JavaScript. Java is more commonly used in production based applications whereas JavaScript is used in web-based programming. This means that publishers can implement a web-based application or a Java application for detecting malicious intent.

The choice of whether to use Java or JavaScript depends on whether the developers want the system to be easily accessible, the programming knowledge of the developers, and their expertise in computer networking. For example, if the user wants a shareable web-based application and they have expertise in web-based programming, networking knowledge, and smart contract programming knowledge, they might opt to go the JavaScript web-based route.



Severity Report

App Name	Severity Level	Fake Click Count	Number of Command-And-Contro Server External IP Address
CalcBlast	Yellow	1	0
Star Catch	Green	0	0
Baseball Blitz	Red	2	1

Green: No counts for fake ad clicks or command-and-control server External IP Addresses

Yellow: App contains at least 1 count for fake ad clicks or command-and-control server

Red: App contains at least 2 count for fake ad clicks or command-and-control server. App contains at least 1 click count for fake clicks and command-and-control server

1. Check if Fake Click Count is greater than zero mark the severity level.
2. Check if External IP Addresses exist for the command-and-control servers.
3. Log the severity level for each application

1. Suppose the Android app has a Fake Click Count of 1. The severity level would be yellow. If the Click count was 2 or greater the severity level would be red.
2. Suppose the Android app has no Fake Click counts, but there is 1 command-and-control External IP address, the severity level would be yellow. If the command-and-control External IP address count was greater than 1, the severity level would be red.

Figure 4. Zen Auditing Framework Overview

5. Evaluation

Using an Ethereum private blockchain was a better solution than a public based blockchain because the public blockchain has a slower transactions rate, and costs are too high. This led to the conclusion of using a private based blockchain. If a publisher was to implement a forensic analysis framework, they would have to use the geth console. The geth console is a tool developed by Ethereum to help create private blockchain environments. Publishers would implement their infrastructure using a private blockchain because it provides better control of the blockchain data and still allows them to connect multiple computers to the same private blockchain.

6. Conclusion and Future Work

This paper illustrates the development of an auditing infrastructure that uses advanced compiler technology and blockchain smart contracts to assist with detecting click fraud. We believe the tool is a powerful and useful tool for advertisement library developers who want to track their applications using a private blockchain.

Injecting a blockchain call into an already packaged Android application is complex, flexible, and useful that companies can leverage to track and monitor applications for security and auditing purposes. Complexity should not deter our commitment to using these tools. Decompilers, forward and backwards flow analysis, code injection, and smart contract technologies are the future of forensic malware analysis.

The limiting factors are the degree of understanding the Soot framework, networking knowledge, malware analysis knowledge, assembly language knowledge, and programming stack knowledge. The Soot framework is particularly useful because it has powerful compiling and decompiling features. It also allows developers and security analyst to provide dynamic and static analysis in their implementations.

One interesting aspect of this research project, is that we have essentially created malware to inject private blockchain smart contract calls into Android applications. The application developer could theoretically create malware to track their Android applications and to send immutable data to the private blockchain. This is a double-edge sword because hackers could of course maliciously use this knowledge to inject malicious code into Android applications without the consent of the user or Android app developer.

Future work could also entail using machine learning approaches along with various analysis techniques to help with detecting whether click fraud exists in an Android application. We believe that k-fold cross validation and gradient boosting have the best potential. Further options will be explored. There are a variety of emerging malware research projects that may be applicable to this research strain. For example, privacy leakage detection [22], malware execution paths [23], behavioral-based malware detection [24], automated detection of botnets [25], and the network analysis of malvertising [26].

An interesting description of the statistics and the various techniques used in mobile malware detection in production environments was introduced by Chandramohan et al. [27]. A good overview of the various mobile malware detection techniques has been developed by Amro et al. [28].

It would also be interesting to explore if it is possible to use dynamic analysis and computer networking to help with detecting advertisement click fraud that occurs from command-and-control servers. In particular we would like to develop a framework that incorporates networking data to identify click fraud that has occurred from the command-and-control servers.

References

[1] "Google Play Store Stats and Facts You Should Know in 2021," July 2019.

[2] "Number of monthly Android app releases worldwide 2021."

[3] R. Samani, "McAfee Mobile Threat Report," report, 2020.

[4] "Blog \textbackslashtextbar Top 10 Malware January 2021," Feb. 2021.

[5] J. D. Vecchio, S. Y. Ko, and L. Ziarek, "Representing string computations as graphs for classifying malware," in *MOBILESoft '20: IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pp. 120–131, ACM. Type: Conference Proceedings.

[6] P. V. Shijo and A. Salim, "Integrated Static and Dynamic Analysis for Malware Detection," *Procedia Computer Science*, vol. 46, pp. 804–811, Jan. 2015.

[7] "Mobile Malware Analysis : Tricks used in Anubis."

[8] L. Li, J. Gao, M. Hurier, P. Kong, T. F. Bissyandé, A. Bartel, J. Klein, and Y. L. Traon, "AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community," *arXiv:1709.05281 [cs]*, Sept. 2017.

[9] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android," Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pp. 86–103, Springer International Publishing. Type: Conference Proceedings.

[10] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," Type: Conference Proceedings.

[11] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "MAST: triage for market-scale mobile malware analysis," WiSec '13, pp. 13–24, Association for Computing Machinery. Type: Conference Proceedings.

[12] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," pp. 32–46. Type: Conference Proceedings.

[13] J. Garcia, M. Hammad, and S. Malek, "Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware," *ACM Transactions on Software Engineering and Methodology*, vol. 26, pp. 11:1–11:29, Jan. 2018.

[14] P. Cook and N. Stakhanova, "Android Malware Classification through Analysis of String Literals," 2016.

[15] Z. Li, J. Sun, Q. Yan, W. Srisa-an, and Y. Tsutano, "Obfusifier: Obfuscation-Resistant Android Malware Detection System," in *Security and Privacy in Communication Networks* (S. Chen, K.-K. R. Choo, X. Fu, W. Lou, and A. Mohaisen, eds.), pp. 214–234, Springer International Publishing. Type: Conference Proceedings.

[16] C. J. Fung, D. Y. Lam, and R. Boutaba, "RevMatch: An efficient and robust decision model for collaborative malware detection," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, pp. 1–9, May 2014.

[17] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. L. Traon, J. Klein, and L. Cavallaro, "Euphony: Harmonious Unification of Cacophonous Anti-Virus Vendor Labels for Android Malware," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 425–435, May 2017.

[18] J. Brownlee, "A Gentle Introduction to Vectors for Machine Learning," Feb. 2018.

[19] P. Beaucamps, I. Gnaedig, and J.-Y. Marion, "Behavior Abstraction in Malware Analysis," in *Runtime Verification* (H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, eds.), Lecture Notes in Computer Science, pp. 168–182, Springer, 2010.

[20] "PHA Family Highlights: Zen and its cousins."

[21] S. Sanders and L. Ziarek, "A comparison and contrast of APKTool and Soot for injecting blockchain calls into Android applications," in *Proceedings of the 54th Hawaii International Conference on System Sciences*, Jan. 2021.

[22] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pp. 1043–1054, Association for Computing Machinery, Nov. 2013.

[23] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, pp. 231–245, May 2007.

[24] I. Firdausi, C. lim, A. Erwin, and A. S. Nugroho, "Analysis of Machine learning Techniques Used in Behavior-Based Malware Detection," in *2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies*, pp. 201–203, Dec. 2010.

[25] J. Tallett, N. Agnese, M. Habiby, C. Soo, and M. LeRoy, "The Shoe is a Lie: How an Android Botnet Defrauded Advertisers and Consumers."

[26] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, "Mobile malware detection through analysis of deviations in application network behavior," *Computers & Security*, vol. 43, pp. 1–18, June 2014.

[27] M. Chandramohan and H. Tan, "Detection of Mobile Malware in the Wild."

[28] B. Amro, "Malware Detection Techniques for Mobile Devices," SSRN Scholarly Paper ID 3430317, Social Science Research Network, Rochester, NY, 2017.

[29] L. Mearian, "What is blockchain? The complete guide," Jan. 2019. Publication Title: Computerworld.

Term	**Definition**
compiler	A program that translates statements written in a source programming language and into machine language, object code or assembly.
decompiler	A program that translates machine language, object code or assembly into a high level language such Java.
bytecode	A low-level representation of program code that has been compiled. It can closely resemble assembly language.
APK	The Android Package Kit is used to distribute and for the subsequent execution of an Android application. It is similar to the exe format in Microsoft Windows.
code injection	The process of injecting statements into an application at a specific location without disturbing the flow of the application code.
soot framework	A compiler framework that is able to decompile and compile Java code with the capability of analysing and instrumenting Java code.
instrumentation	Refers to the modification and analysis of a programming language through the use of compiler technology.
jimple	An intermediate representation of Java code that Soot generates as output.
blockchain	A peer-to-peer network that allows for the sharing of data among a vast number of peers [29]. All data stored on the blockchain is immutable.
Ethereum blockchain	A blockchain environment that allows the use of smart contracts.
smart contract	A contract with written rules and terms allowing for controlling the storage, sharing, and modification of data.
Ganache	A tool used for creating an Ethereum blockchain environment.
solidity	A smart contract object-oriented programming language that was developed by Ethereum.
Remix	Ethereum's tool that helps developers program smart contracts. It enables smart contract developers to connect and push smart contracts to the Ethereum blockchain.
DApps	This refers to the decentralized, resilient, transparent, and incentivized applications that reside on blockchain infrastructures. These applications are supposedly less prone to errors.
Backward flow analysis	Provides information about the future code along the path of execution.
Forward flow analysis	Provides information about past code along the path of execution.
Malvertising	Android advertising malware (malvertising) involves fraudulent behavior related to advertising libraries.

Table 1. Terminology