Progressive Control Flow Obfuscation for Android Applications

Li Zhang, Huasong Meng, Vrizlynn L. L. Thing
Institute for Infocomm Research
A*STAR
Singapore
{zhang-li, menghs, vriz}@i2r.a-star.edu.sg

***Abstract*—Android bytecode is easy to reverse engineer. It has been a common practice for Android application developers to protect their applications with obfuscation techniques. Control flow obfuscation aims to make it more difficult to determine the actual application control flows and thereby impede the understanding of the application logic by the attacker. Despite of the strong potency (i.e., high complexity increment), control flow obfuscation usually incurs a large overhead due to the call and return instructions inserted, which makes the application developer reluctant to use it in practice. In this paper, we present a pragmatic control-flow obfuscation approach where the application developer has more freedom to customize the trade-off between the achieved complexity and overhead. A new subset of application methods will be obfuscated by using a combination of packed-switch and try-catch constructs in different rounds, and larger methods are obfuscated by creating more code fragments in earlier rounds. After each round, the complexity increment will be automatically calculated using our implemented cyclomatic complexity based metric and checked against the target complexity increment. In other words, the obfuscation is conducted in a progressive manner until the target complexity increment is reached. The experimental results show that our method incurs averaged area overhead of 4.07% while achieving almost double complexity increment than the existing method when the same number of application methods are obfuscated.**

***Keywords*—Android, application security, software obfuscation, reverse engineering**

I. Introduction

Smart phones have been pervasive in our daily life. People are increasingly relying on smart phone applications for social networking, media streaming, financial transactions, etc. In the year 2017 alone, a total of 1.5 billion units of smart phones were purchased by the end users, among which 86% of devices were powered by Android [1]. At the same time, the revenue generated by Google Play and third-party Android application stores reached $41 billion in 2017 and is projected to be $78 billion in 2021 [2]. The extensive user base and the easy-to-reverse nature of Android applications has, however, made these applications the most attractive targets for cybercriminals. For example, an attacker may obtain the code of a popular Android application through reverse engineering, modify the code to remove copyright information or add malicious modules, and then repackage and release the application under his own key.

Android applications are important intellectual property of the original developers. To protect them from possible attacks, it has been common for the developers to use obfuscation techniques. Obfuscation is the process of obscuring the program attributes that can be potentially exploited by reverse engineering while preserving the semantics of the original code. There have been a number of Android obfuscation tools available in the market, such as Proguard [3], DexGuard [4], DexProtector [5], and DashO [6]. These tools offer multiple layout or data obfuscation options such as identifier renaming, debugging information removal, and encryption of strings, arithmetic instructions, and resources. Some of these tools can further perform simple control flow obfuscation by inserting opaque predicates [7] (i.e., conditional expressions that always returns *true* or *false* but are hard to be determined by static analysis). However, none of these tools offer complex control-flow obfuscation like those proposed for other platforms [8–10], which can scatter code fragments in application methods and increase the control flow diversion among the fragments.

There are also a limited number of complex control flow obfuscation methods for Android applications found in literature. Although the various control-flow obfuscation techniques proposed at the Java source code level [11–13] are also applicable to Android applications, due to the problems of variable scoping and data dependency, they share the same drawback that code splitting can only be done at the boundaries of the basic blocks [8]. An additional challenge for performing control flow obfuscation at the Java source code level comes from the lack of unconditional jump instructions like `goto`. As a result, it is hard to transform high-level code constructs like `while` and `for` into the simple `if-then-goto` constructs for code splitting. The work in [14] proposed a Java bytecode based control flow obfuscation technique, where the Java bytecode is firstly translated to an intermediate representation (IR) called Jimple [15] and `goto` instructions are then inserted to facilitate code splitting. Nonetheless, the obfuscator targets the Java bytecode executed in the stack-based JVM, and cannot be directly applied on Android bytecode executed in the register-based Android virtual machine (i.e., Dalvik virtual machine (DVM) or Android runtime (ART)).

This material is based on research work supported by the Singapore National Research Foundation under NCR Award No. NRF2014NCR-NCR001-034.

Simple control flow obfuscation techniques based on the insertion of opaque predicates were applied on Android applications in [16, 17] to examine the effectiveness of Android anti-malware systems in detecting transformed malware variants. The first complex control flow obfuscation method for Android applications was proposed in [18], where the control flow of each obfuscated application method is disturbed by inserting a combination of the `packed-switch` and `try-catch` constructs as well as `goto` instructions into the smali representation of the Android bytecode. To resolve the possible register-type conflict issue, which may be raised by the Android virtual machine when the bytecode is split or relocated, a register-type separation technique was used. In the approach, however, all obfuscated application methods are split into the same number of code fragments, which results in a fixed and constant complexity increment for each obfuscated method. In fact, as the size of application methods ranges from several lines of code (LoC) to hundreds of lines, the number of possible code fragments that can be generated (i.e., the potential complexity increment for these methods) is different. In view of this, we propose a new progressive obfuscation approach, where larger methods in the application are obfuscated by creating more code fragments in earlier rounds. Besides, unlike the method in [18] which just qualitatively analyze the complexity increment, we implement a cyclomatic complexity based metric and use it to quantitatively measure the achieved complexity increment after each round of obfuscation. There are two obvious advantages for the new approach. Firstly, higher complexity increment can be achieved when the same number of application methods are obfuscated (at a reasonably low area overhead, as will be shown in section III). Secondly, as the obfuscation process is divided into several rounds and performed in a progressive manner, by utilizing the quantified complexity increment, the user is able to decide whether the target complexity increment is achieved after each round of obfuscation. This will facilitate a good balance between the protection level and overhead.

The primary contributions of this work are as follows:

- We propose a new progressive approach to obfuscate the control flow of Android application methods. The obfuscation process is divided into several rounds, where larger application methods are obfuscated in earlier rounds with more code fragments created. Compared to the existing method which creates a constant number of code fragments for each obfuscated method, our new approach better exploits the potential of complexity increment of the obfuscated methods.
- We propose to use a metric to quantify the complexity increment during the obfuscation process. With the implemented cyclomatic complexity based metric, the user is able to decide whether the complexity increment has reached his desired level after each round of obfuscation. As a result, the application developer will be able to customize the trade-off between the resultant complexity and overhead.

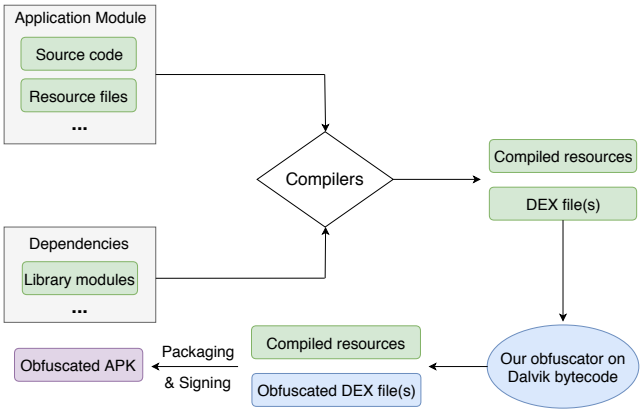

Fig. 1: The build process of obfuscated Android application in our method.

II. The Proposed Method

Android applications are mainly written in Java codes, which are then compiled along with data and resource files into an archive file called Android Package Kit (APK). The binary Android bytecode file (.dex) in the APK contains all the classes and can be converted to an intermediate representation called smali code (akin to assembly code). The smali code of an application method is the text-based mnemonics equivalent to the method bytecode. This kind of low-level representation can express arbitrary control flow due to the existence of `goto` instructions (missing in Java code) [11]. As a result, we choose to implement our control flow obfuscation method on the smali representation of Dalvik bytecode.

Figure 1 shows the process of building an obfuscated application in our method. The obfuscation is transparent to the typical build process of a normal Android application. Our obfuscation method consists of two main phases. In the first phase, statistics about the number of application methods that can be divided under different numbers of code fragments are extracted. This phase facilitate the determination of the different numbers of code fragments to be used in obfuscation. In the second phase, the multi-round based progressive obfuscation is conducted. At each round, a new subset of application methods will be obfuscated based on the corresponding number of code fragments. Details of these two phases are described in the remaining part of this section.

A. Statistical analysis of the target application

Basically, our technique achieves method level obfuscation. It obfuscates the control flow of an application method by splitting its code into multiple code fragments and shuffling their respective locations while preserving the original execution sequence. We denote the number of code fragments as n_c. The size of the application method (i.e., in terms of LoC) determines the maximum n_c it can be divided. Nonetheless, the Android virtual machine, in which the application is executed, specifies various static and structural constraints for the bytecode to ensure the runnability of the application [19]. For instance, the instruction `move-result*` must be immediately preceded by a method invocation instruction `invoke-*`.

As a result, we have to make sure these two instructions stay in the same code fragments after code splitting. Due to these constraints, the maximum n_c for a specific application method is further reduced. In our method, the smali representation of the application is firstly input to a preprocessing module. The preprocessing module, which loads the constraints that must be met to maintain the application runnability, scans the application to determine the maximum n_c for each method. The output of this module is the statistics for the number of application methods that can be obfuscated under different n_c. Such statistics facilitate the determination of the list of n_c's (denoted as $\overrightarrow{n_c}$) that will be used in the following multi-round based obfuscation phase. As a rule of thumb, the list $\overrightarrow{n_c}$ should be chosen such that a good number of methods are obfuscated in each round.

B. The multi-round based progressive obfuscation

The obfuscation phase consists of multiple rounds. Suppose that based on the statistics obtained in the preprocessing step, the user decides to perform *m* rounds of obfuscation, where $\overrightarrow{n_c} = \{n_{c_1}, n_{c_2}, \ldots, n_{c_m}\}$, $|\overrightarrow{n_c}| = m$, and $n_{c_i} > n_{c_j}$ if $i < j$. In round *i*, the corresponding subset of application methods that can be obfuscated under n_{c_i} ($i \in [1, m]$) but have not been obfuscated in previous rounds will be obfuscated. After obfuscating all the methods at round *i*, the complexity increment is measured by a cyclomatic complexity based metric. The cyclomatic complexity measures the total number of paths in the control flow graph (CFG) of the target method, and can be calculated based on the formula $e - n + 2$ [20], where *e* and *n* represents the edges and nodes of the method CFG, respectively. Based on the complexity increment at the end of each round of obfuscation, the user can decide whether the overall complexity increment is good enough to stop the obfuscation. If yes, he may choose to obscure the control flow of the remaining application methods by simply inserting opaque predicates. Or else, the obfuscation will continue with a new round by processing a new subsets of application methods with the smaller $n_{c_{i+1}}$. By obfuscating the methods which can be divided into more code fragments, a faster complexity increment per obfuscated method is achieved. The pseudocode for the proposed progressive obfuscation approach is presented in Figure 2.

To obfuscate a specific method, we utilize the technique proposed in [18], which is based on the use of the `packed-switch` and `try-catch` instructions. Firstly, the target method is split into n_{c_i} ($i \in [1, m]$) code fragments, where the constraints imposed by Android virtual machine are considered. Due to the availability of unconditional `goto` instruction in the smali representation, the code splitting can be performed within a basic block. Then the code fragment locations in a method are randomly shuffled to disrupt the linear order. To ensure that the code fragments are executed as per the original order during runtime, the `packed-switch` instruction within a loop construct is used, together with a new register directing the transfer between the code fragments. Afterwards, the entire obfuscated code segment is put

Input:
$\overrightarrow{M_T}$ # all methods in the target app
$\Delta_{desired}$ # desired complexity increment after obfuscation (optional)
$\overrightarrow{n_c}$: # the list of n_c to be used in each round of obfuscation
$|\overrightarrow{n_c}| = m$ and $n_{c_i} > n_{c_j}$, if $i < j$
Output:
The obfuscated app

$\overrightarrow{M_{obf}} = \emptyset$ # initialize an empty list of obfuscated methods
$\Delta = 0$ # total complexity increment, initialized to 0
for i = 1, i <= m, i++:
Collect $\overrightarrow{M_i}$ # $\overrightarrow{M_i}$ is a subset of $\overrightarrow{M_T}$ obfuscatable at n_{c_i}
$\overrightarrow{M_i} \cap \overrightarrow{M_{obf}} = \emptyset$

for M_j **in** $\overrightarrow{M_i}$: # obfuscate all methods in $\overrightarrow{M_i}$
Obfuscate M_j
$\overrightarrow{M_{obf}} = \overrightarrow{M_{obf}} \cup \overrightarrow{M_i}$ # mark all methods in $\overrightarrow{M_i}$ as obfuscated
Δ += measured complexity increment for $\overrightarrow{M_i}$
if $\Delta_{desired}$!= None **and** Δ >= $\Delta_{desired}$:
break

Fig. 2: Pseudocode for the proposed propressive obfuscation approach.

inside a `try` block, where the `packed-switch` instruction is replaced by an exception-raising instruction and itself is moved from the `try` block to the `catch` block. The added `try-catch` based obfuscation helps further divert the control flow between the `try` and `catch` block. An example of the obfuscated code is shown in Figure 3. For the simplicity of illustration, the switching register v1 is directly assigned with values (during both initialization and updates in the code fragments) and a simple division-by-zero exception is used. In practice, we can use the opaque variable whose value is generated using a hash function, which makes it much more difficult to infer the values by static analysis.

III. Experimental Evaluation and Discussion

The smali representation of the Dalvik bytecode was generated using apktool [21]. We implemented the constraints from the Android virtual machine in both the preprocessing module and the obfuscation module to make sure both the statistical analysis and obfuscation are performed at the condition of maintaining the runnability of the application. To implement the cyclomatic complexity metric, we used the simple way suggested in [20], which is based on counting the number of code constructs indicating decision predicates. To evaluate the overhead and efficacy of our new obfuscation approach, we leveraged the same experimental setup as in [18], where the same five popular applications downloaded from the Google Play store were used. Information about these five applications such as the name, size, number of methods, and original complexity of all methods are presented in Table I.

Our method achieves the target of obfuscating the control flow while preserving the application semantics with the aid of inserted instructions such as `packed-switch`, `try-catch`, and `goto`. To evaluate the possible impact on the size of the resultant apk file, we firstly perform statistical analysis on the five applications using the preprocessing module to extract the number of application methods obfuscatable

```
const v1, 0

:try_start_obf
  :loop
  # a simple division-by-zero exception
  const v3, 0x5
  div-int/lit8 v3, v3, 0x0

  # randomly shuffled n code fragments - start
  :obf_pswitch_0
    <code_fragment_0>
    # two instructions added to direct the code execution
    const v1, 3
    goto :loop

    ...

  :obf_pswitch_n
    <code_fragment_n>
    # two instructions added to direct the code execution
    const v1, 5
    goto :loop
  # randomly shuffled n code fragments - end

  # packed swtich payload block
  :pswitch_obf
    .packed-switch 0x0
      :obf_pswitch_0
        ...
      :obf_pswitch_n
    .end packed-switch
:try_end_obf

.catch Ljava/lang/ArithmeticException;
  {:try_start_obf .. :try_end_obf} :catch_obf

:catch_obf
  packed-switch v1, :pswitch_obf
```

Fig. 3: An example smali code obfuscated with the `packed-switch` and `try-catch` instruction. The register v1 is used as the flag register to transfer the execution between the *n* code fragments.

TABLE I: Information of the five applications used in the experiments.

App name	Size (KB)	# methods	Complexity
Winamp 1.4.15	7,572	10,527	16,908
Wechat 4.5.1	18,356	38,262	56,697
Line 3.8.4	16,128	43,242	69,740
Sound cloud 2.8.3	7,540	26,645	29,381
Photo editor 1.3.18	1,988	7,412	10,940

at different n_c. Based on the statistics, we decided to obfuscate all the 5 applications based on the list of $\overrightarrow{n_c} = [15, 12, 9, 6, 3]$. Under this list of $\overrightarrow{n_c}$, there will be at least 150 application methods that can be obfuscated at each round. This means that the methods that can be divided into 15 code fragments (without breaking any runtime constraints from the Android virtual machine) were obfuscated in the 1st round, then the remaining methods obfuscatable with 12 code fragments in the 2nd round, and so on. The resultant size overhead together with the number of methods obfuscated in each round is shown in Table II. As expected, the size of the application generally grows with the number of obfuscated methods. Nonetheless, after all 5 rounds of obfuscation, where on average 30% of all application methods were obfuscated, the averaged percentage of size increase is just 4.07%.

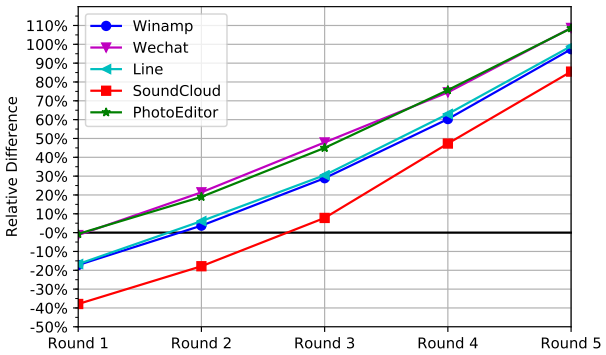

Fig. 4: The relative difference of total complexity increment achieved in [18] and after each round of obfuscation in our method.

A major advantage of the proposed method compared to [18] is that it can achieve much higher complexity by obfuscating the same number of methods. The comparison of the complexity increment is presented in Table III. The 2nd column shows the number of obfuscated methods for each application using the technique in [18], where all obfuscated methods were divided into 3 code fragments (i.e., under $n_c = 3$). This number is equal to the total number of obfuscated methods in 5 rounds using our method (see Table II). It can be observed that the overall complexity increment using the method in [18] and our method (after 5 rounds of obfuscation) was around 100% and 200% for all applications, respectively. Figure 4 shows relative difference between the total complexity increment achieved in [18] and after each round of obfuscation using our approach. It is clear that after three rounds, our method has already achieved higher complexity increment for all applications. In fact, for all applications except for Sound cloud, higher complexity increment was even achieved after the 2nd round. The reason for Sound cloud having slower complexity increment should be due to the smaller percentage of large methods. As shown in Table II, the number of obfuscated methods in the first two rounds accounts for around 25% of all obfuscated methods for Sound cloud, while the percentage for the other four applications ranges from 32% to 37%. Hence, the percentage of larger methods in an application will impact the effectiveness of our progressive obfuscation approach. Nonetheless, for all applications, we just need to obfuscate less than $1/3$ of the methods used in [18] to achieve the same complexity increment.

On the other hand, additional complexity can be easily achieved by slightly modifying our obfuscation process. For instance, after splitting the target application method into n_c code fragments, multiple junk code fragments can be inserted. As there will be edges for these junk code fragments in the reverse-engineered CFG, the attacker will be further confused. As the flag register is not assigned the values needed to execute these fragments, they will not impact the actual execution of the application. Similarly, in the created `try` block, junk codes can also be inserted after the exception-raising operations.

TABLE II: Size overhead of the proposed progressive obfuscation approach. All application sizes are in KB. Column *#obfuscated method* lists the number of methods obfuscated at the specific round, while Column *Size after obfuscation* lists the resultant APK size if obfuscation stops at the round.

	Round 1($n_c = 15$)		Round 2 ($n_c = 12$)		Round 3 ($n_c = 9$)		Round 4 ($n_c = 6$)		Round 5 ($n_c = 3$)	
App name	# obfuscated method	Size after obfuscation	# obfuscated method	Size after obfuscation	# obfuscated method	Size after obfuscation	# obfuscated method	Size after obfuscation	# obfuscated method	Size after obfuscation
Winamp	715	7,652 (+1.06%)	219	7,672 (+1.32%)	337	7,688 (+1.53%)	575	7,708 (+1.80%)	1,090	7,728 (+2.06%)
Wechat	3,877	18776 (+2.29%)	1,082	18,852 (+2.70%)	1,606	18,924 (+3.09%)	2,219	18,996 (+3.49%)	4,566	19,072 (+3.90%)
Line	3,262	16440 (+1.93%)	1,086	16,508 (+2.36%)	1,475	16,584 (+2.83%)	2,700	16,668 (+3.35%)	4,794	16,752 (+3.87%)
Sound cloud	1,100	7664 (+1.64%)	429	7,692 (+2.02%)	704	7,732 (+2.55%)	1,488	7,784 (+3.24%)	2,301	7,832 (+3.87%)
Photo editor	629	2060 (+3.62%)	154	2,076 (+4.43%)	252	2,088 (+5.03%)	413	2,104 (5.84%)	706	2,120 (+6.64%)
Average		+2.11%		+2.57%		+3.01%		+3.54%		4.07%

TABLE III: Comparison of complexity increment with our progressive obfuscation approach and the one in [18] (adopting $n_c = 3$ for all methods). Column *complexity increment* lists the total complexity increment and percentage difference of the resultant application compared to that of the original application.

	Method proposed in [18] ($n_c = 3$)		Round 1 ($n_c = 15$)	Round 2 ($n_c = 12$)	Round 3 ($n_c = 9$)	Round 4 ($n_c = 6$)	Round 5 ($n_c = 3$)
App name	# obfuscated methods	Complexity increment	Complexity increment	Complexity increment	Complexity increment	Complexity increment	Complexity increment
Winamp	2,936	14,680 (+86.82%)	12,155 (+71.89%)	15,221 (+90.02%)	18,928 (+111.94%)	23,528 (+139.15%)	28,978 (+171.39%)
Wechat	13,350	66,750 (+117.73%)	65,909 (+116.24%)	81,057 (+142.97%)	98,723 (+174.12%)	116,475 (+205.43%)	139,305 (+245.70%)
Line	13,317	66,585 (+95.48%)	55,454 (+79.52%)	70,658 (+101.32%)	86,883 (+124.58%)	108,483 (+155.55%)	132,453 (+189.92%)
Sound cloud	6,022	30,110 (+102.48%)	18,700 (+63.65%)	24,706 (+84.09%)	32,450 (+110.45%)	44,354 (+150.96%)	55,869 (+190.12%)
Photo editor	2,154	10,770 (+98.45%)	10,693 (+97.74%)	12,849 (+117.45%)	15,621 (+142.79%)	18,925 (+172.99%)	22,455 (+205.26%)

IV. Conclusion

We proposed a progressive approach to obscure the control flow of Android applications, where larger methods are divided into more code fragments and obfuscated in earlier rounds of obfuscation. Statistics on the number of methods that can be obfuscated at different numbers of code fragments were extracted to assist the obfuscation process. After each round, the cyclomatic complexity based metric will quantitatively measure the complexity increment so that the user can decide whether the obfuscation has already met his requirement, hence providing more freedom to balance the obfuscation complexity and overhead. Despite of achieving higher complexity increment than the existing method, our method just incurs reasonably low overhead, which makes it feasible to use our method together with other types of obfuscation techniques to better safeguard the IP of the application developer.

References

[1] Gartner. Gartner says worldwide sales of smartphones recorded first ever decline during the fourth quarter of 2017. [Online]. Available: https://www.gartner.com/newsroom/id/3859963.

[2] Bussiness of Apps. App revenues 2017. [Online]. Available: http://www.businessofapps.com/data/app-revenues/.

[3] ProGuard. [Online]. Available: http://developer.android.com/tools/help/proguard.html.

[4] DexGuard. [Online]. Available: https://www.guardsquare.com/software/dexguard-standard.

[5] DexProtector. [Online]. Available: https://dexprotector.com.

[6] DashO. [Online]. Available: https://www.preemptive.com/products/dasho.

[7] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998, pp. 184–196.

[8] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software tamper resistance: Obstructing static analysis of programs," Technical Report CS-2000-12, University of Virginia, 12 2000, Tech. Rep., 2000.

[9] S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov, "An approach to the obfuscation of control-flow of sequential computer programs," in *International Conference on Information Security*. Springer, 2001, pp. 144–155.

[10] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals." in *USENIX Security Symposium*, 2007, pp. 275–290.

[11] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[12] D. Low, "Protecting java code via code obfuscation," *Crossroads*, vol. 4, no. 3, pp. 21–23, 1998.

[13] ——, "Java control flow obfuscation," Ph.D. dissertation, Citeseer, 1998.

[14] M. Batchelder and L. Hendren, "Obfuscating java: The most pain for the least gain," in *International Conference on Compiler Construction*. Springer, 2007, pp. 96–110.

[15] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*. IBM Corp., 2010, pp. 214–224.

[16] V. Rastogi, Y. Chen, and X. Jiang, "Catch me if you can: Evaluating android anti-malware against transformation attacks," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 1, pp. 99–108, 2014.

[17] M. Zheng, P. P. Lee, and J. C. Lui, "Adam: an automatic and extensible platform to stress test android anti-virus systems," in *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 2012, pp. 82–101.

[18] V. Balachandran, D. J. Tan, V. L. Thing *et al.*, "Control flow obfuscation for android applications," *Computers & Security*, vol. 61, pp. 72–93, 2016.

[19] Android open source project - constraints. [Online]. Available: https://source.android.com/devices/tech/dalvik/constraints.

[20] A. H. Watson, D. R. Wallace, and T. J. McCabe, *Structured testing: A testing methodology using the cyclomatic complexity metric*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1996, vol. 500, no. 235.

[21] APK Tool. A tool for reverse engineering android apk files. [Online]. Available: https://ibotpeaches.github.io/Apktool/.