Don't Trust Me, Test Me: 100% Code Coverage for a 3rd-party Android App

Aleksandr Pilgun
SnT, University of Luxembourg
Belval, Luxembourg
aleksandr.pilgun@uni.lu

***Abstract*—The incompleteness of 3rd-party app testing is an accepted fact in Software Engineering. This issue makes it impossible to verify the app functionality and to confirm its safety to the end-user. To solve this problem, enterprises developed strict policies. A company, willing to use modern apps, may perform an expensive security analysis, rely on trust or forbid the app. These strategies may lead companies to high direct and indirect spending with no guarantee of safety.**

In this work, we present a novel approach, called Dynamic Binary Shrinking, that allows a user to review app functionality and leave only tested code. The shrunk app produces 100% instruction coverage on observed behaviors and in this way guarantees the absence of unexplored, and therefore, potentially malicious code. On our running examples, we demonstrate that apps use less than 20% of the codebase. We developed an approach and the ACVCut tool to shrink Android apps towards the executed code.

Repository — http://github.com/pilgun/acvcut [1].

I. Introduction

Over more than a decade, a joint effort of industry and academia helped Android and its ecosystem evolve towards a safer platform [2]. Yet, given its 86%[1] in market share, it remains a precious target for attackers. The security measures introduced into the Android ecosystem drive the system to be more secure. However, attacks also evolved into more sophisticated and targeted [3], [4].

One of the big achievements Google recently fulfilled is the certification of Google Pixel 3/3XL devices as compliant to Common Criteria[2]. Certification allowed strict-policy enterprises and government institutions to use Google phones trustfully. Besides devices, the document certified Android 9.0, mentioned new libraries and rich APIs for app development.

However, 3rd-party apps are not the target of evaluation in this certification. The enterprise version of Android has a Mobile Device Management (MDM) system capable of remote control of apps and phone settings [5]. However, vulnerabilities and developed attacks continue to threaten Android ecosystem [3], [4], [6].

Logic bombs are a great example of a popular targeted attack. For example, recently, in the wake of increased interest in cryptocurrencies, the Google Play app market has filled up with cryptojacking — apps mine cryptocurrencies secretly from the device owner [7]. Attackers more actively used devices at full capacity during the owner's inactive hours and when charging. In a more severe scenario, a logic bomb-driven attack may target people with certain powers or secrets up to state-sponsored attacks [4].

Researchers and practitioners developed many approaches for analyzing Android apps [8], [9]. However, today there is no approach that can guarantee security of all third-party Android apps. Currently, sandboxing [10], [11] is a solution on the frontier to restrict not observed functionality. Still, this approach continues evolving [11], [12] but so far, it does not provide a simple convincing metric that may guarantee safety from hidden behaviors. Android apps need a simple, ready-to-go solution to verify existing behaviors, restrict not observed potentially malicious ones, and give a guarantee with a simple metric. A more radical path we propose in this work is the removal of not tested code from the app.

We present a novel approach, called Dynamic Binary Shrinking. It allows to monitor app behavior while exploring the app and eventually cut it towards the tested benign code. In this way our approach guarantees absence of potentially malicious code with 100% instruction coverage.

The contributions of our work are highlighted as follows:

- We propose Dynamic Binary Shrinking System — a novel methodology created to shrink 3rd-party Android apps towards observed benign functionality on executed code.
- We present a sophisticated technique for removal of not executed code from 3rd-party Android apps.
- We incorporate the presented technique into an open-source tool ACVCut, allowing anyone to shrink 3rd-party apps towards only executed code.
- On running examples, we demonstrate that apps tend to keep significant amounts of not used code. More than 80% of apps code stayed not executed in our samples.
- We demonstrate the viability of our shrinking technique on running examples.

In the rest of the paper, we expand on the subject as follows. Section II gives a background overview. We present our running examples in Section III. We describe the proposed methodology in Section IV. We give a detailed description of our code modification technique in Section V. Section VI applies shrinking on the running examples. Section VII discusses threats, limitations, and future work to address them. Section VIII is for related work. We conclude in Section IX.

[1] https://www.idc.com/promo/smartphone-market-share/os
[2] https://www.niap-ccevs.org/Product/Compliant.cfm?PID=10941

II. Background

Android apps is a complete software product written mainly in Java for Android OS. Android users may find apps created by 3rd-party developers in Google Play and other app stores, such as Amazon Appstore and F-Droid to mention a few. Alternatively, a user may copy or download an app anywhere on the Internet and install it himself.

The APK file (Android package) consists of various resource files, one or more compressed binary executable (DEX) files, AndroidManifest.xml (declares app capabilities) and native C/C++ libraries often used for performance-sensitive operations.

Android provides developers with a rich application framework for faster app development and backward compatibility to different Android versions. Also, developers may use 3rd-party libraries of their choice.

When published, the app undergoes Google Play scans for malware. However, Google Play can not guarantee the safety of available apps, and therefore, adds more protection on the user site [2].

A. Security enforcement on Android devices

An app communicates with Android OS through the application framework to perform various actions using Android APIs. Sensitive Android APIs cover such functions as obtaining user location or sending an SMS and stay behind a user-controlled permission mechanism. At any time, the user can grant or take away the permission from the app.

On Android, sensitive APIs restricted by permissions is one of the main targets to both defenders and attackers [4], [13]. Indeed, early works focused on enforcing correct permissions usage in apps [14]. Although permissions may be granted correctly, an attacker may use benign functionality declared in the app for malicious purposes. One of the mechanisms is to inject a logic bomb — malicious code that triggers only under peculiar conditions. Such code is, therefore, invisible under normal usage [4].

One approach to enforce user safety from malicious apps comes from the idea *to turn the incompleteness of dynamic analysis into a guarantee* [11], [15] and offers a sandbox that restricts the app towards only explored behaviors. Another approach aims to remove the dead code (and decrease the attack surface) using static analysis [16]–[19] or dynamically measuring traces of executed code [20]–[22].

B. Program analysis

Researchers and practitioners continue improving static and dynamic techniques to identify malware. Both techniques however have their challenges, e.g. static analysis suffers from *overapproximation* while dynamic analysis — from *incompleteness* of testing [15]. Many researchers believe that hybrid analysis combining both static and dynamic benefits is more efficient [23].

Static analysis of 3rd-party Android apps usually relies on Soot [24] framework, which converts app binaries into Jimple intermediate representation and provides vast capabilities for code analysis. Several approaches use Soot and Jimple to cut excess code from the apps [16]–[18]

`Smali` is another intermediate representation for Android apps. It is supported by Google [25] and precisely reflects Android bytecode instructions into `smali` instructions making app bytecode human readable. Hence, `smali` representation gained significant popularity when reverse engineering, repackaging and hacking Android apps.

The goal of dynamic analysis is to report on the activities the app performed when running. A number of automated testing tools such as DroidMate [10], Droidmate-2 [26], Droidbot [27] and simple Monkey [28] emerged to automate testing routines.

III. Running Examples

In this section, we study the behaviors of two apps. One of them is a toy example where we purposely injected a hidden time bomb. Another one is the real-life Twitter Lite app. We explored both apps to find out how much of the code is executing in the app. We further give present the interpretation of our measurements.

A. Time Bomb Sample

The goal of the Time Bomb sample is twofold. First, it gives us an understanding of the exact amount of code run under a complete test suite. Second, further, in Section VI, we demonstrate in detail how our approach eliminates hidden, potentially malicious functionality out of the app.

We based our sample on the available at Github Java example of a time bomb [29]. To build the app, we used Android Studio 3.5.2, build tools 29.0.2 with targeted minimum Andriod SDK 25, and enabled androidx[3]. We also enabled all available optimizations to minify extra code as much as possible. The optimizations included Proguard [30] resource shrinking, code optimization, and minification and R8 aggressive optimizations in the full mode [31]. Optimizations helped to drop the APK size from 1328KB to 784KB (41% smaller), while the compressed binary code (DEX) file located inside the APK decreased from 1879KB to 262KB (86% smaller).

Indeed, our app is straightforward. It contains only one activity and a timer configured for one day from the app's build time. When launched, the app displays a text indicating one day left before the time expires (*day 0* functionality). However, the next day, when the timer is over, the app triggers an alert (*day 1* functionality). We further use the terms *day 0* and *day 1* to specify the app states before and after the time bomb explosion respectively. We also declare *day 1* functionality as hidden, since it stays unexpected after exploration on the *day 0*. Thus, we have an optimized time bomb app that explodes on the *day 1*.

1) Exploration: Since our app has no buttons or other elements to interact with, and we are fully aware of its functionality, we developed a short but complete test routine intending to achieve peak coverage. Our test suite combines

[3]https://developer.android.com/jetpack/androidx

the following set of actions: launching the app, hiding the app and bringing it to front, switching between the apps, changing a device orientation, stopping the app with a close button, stopping it with a swipe and with the *Clear all* button, launching the app again, taking a screenshot, changing volume, tapping, swiping. Though we did not declare most of the listed actions in the app, Android Application Framework libraries may have default listeners to some of them. In this scenario, an attacker targets to demonstrate benign behavior and pass our testing routine on *day 0*. We now report instruction coverage measured in testing on *day 0* and *day 1*.

When testing the app, we measured instruction coverage with the help of ACVTool [32], [33]. It provides us with information on exact executed instructions (this becomes essential in Section V). ACVTool measures instruction coverage based on `smali` representation of the app's bytecode. It first instruments the app and then tracks executed instructions at the run-time. After the end of exploration, it generates an instruction coverage report. We improved the tool by adding the possibility to measure code coverage at any time without stopping the exploratory procedure.

In our experiments we managed to reach instruction coverage peak with 15.22% and 14.94% on *day 0* and *day 1* respectively. Remarkably, the app demonstrated lower code coverage on *day 1*, though the main package showed the opposite picture — instruction coverage is higher on *day 0* with 52.63% against 72.37% on *day 1*. Furthermore, we observed more UI elements on *day 1* because the time bomb functionality runs an alert on top of the existing page.

However, a closer look at coverage differences revealed that on *day 1* some methods from Android Application Framework libraries did not run compared to *day 1*. These methods are `dispatchKeyEvent`, `dispatchTouchEvent`, `onWindowFocusChanged`, `dispatchKeyEvent`, `onKeyDown`, `setBackgroundDrawable`, `onBackPressed`. This happened due to the time bomb alert message popped out over the main page and blocked default event listeners on the page. We could not identify some other framework methods due to their obfuscation. The difference in the main package only revealed two different branches that executed with regard to the current time.

Although we revealed the hidden feature on *day 1*, the observer could not predict it from *day 0*. Moreover, more than 80% of the app code stayed not tested and we (as app developers) could not guarantee the absence of another logic bomb in this code.

B. Twitter Lite App

The second, more complex example, we took a real-life case — a popular social network app Twitter Lite (version 2.1.2). This is the secondary official Twitter client that targets lower-cost devices limited on storage and performance. Compared to the main official Twitter app, the Lite version has a much smaller size: 1.3Mb against 31.8Mb of the main app. Twitter Lite is a popular app with over 10 million installs and 4 stars rating at Google Play Market. This app is convenient for us due to its relatively generalized and straightforward architecture. In particular, its architecture simplifies our exploratory procedure in an attempt to reach peak instruction coverage. We further describe Twitter Lite features and our results on observing its behaviors.

1) App internals: Twitter Lite is a typical example of a lightweight client app for network communication. Along with Twitter, other major companies, targeting emerging markets, provide Lite versions of apps to their users. To mention a few popular, Facebook Lite, Messenger Lite, Instagram Lite (temporarily rolled back to date), LINE Lite, Skype Lite, Pinterest Lite.

Lite apps make use of a single trick to achieve lightness. An app embeds a web browser that loads and displays web pages from the dedicated website. The embedded browser has no regular address field nor navigation buttons. Thus, the app is often indistinguishable from an ordinary native Android app.

Twitter, as well as most of the other Lite apps, runs the standard Android component WebView [34] to vest the app with browser capabilities. This component allows Twitter to display web page content through the embedded browser, but also it provides JavaScript APIs to interact between the web page, app code, and Android system. WebView often simplifies developers' work because they can adapt original website pages to fit in a mobile app without significant changes in their website logic. This may reduce development costs compared to native apps. Second, it is possible to instantly roll out updates for the app on the website without undergoing the whole update procedure in the app market. However, the main advantage when using WebView is the small size of the app.

In our analysis, we found that the app was compiled using a standard Android `dx` compiler, but the code was obfuscated and minified (most likely, Proguard [30] was used). The app makes a check if a debugger is connected and collects device information. Developers often use such techniques for anti-debugging purposes [35]. In addition to the standard Android Framework libraries, Twitter Lite also uses such popular libraries as retrofit2, okhttp3, crashlytics, firebase, and gson. AndroidManifest file declares permissions from several categories. In the Photos/Media/Files and Storage permission categories: read, modify, and delete the contents of USB storage. From the "Other" category the app declares permissions to receive data from the Internet, view network connections, obtain full network access, run at startup, and prevent the device from sleeping[4].

2) Functionality: The Twitter Lite app's major features include posting a tweet, finding and following another account, bookmarks, home timeline, notifications, profile settings, and direct messages[5].

This functionality is available through the embedded WebView that takes full display space. Therefore, a user mostly

[4]Google Play demonstrates required permissions at the app's page as well: https://play.google.com/store/apps/details?id=com.twitter.android.lite.

[5]Twitter hosts a detailed description of Twitter Lite functionality on its website: https://help.twitter.com/en/using-twitter/twitter-lite.

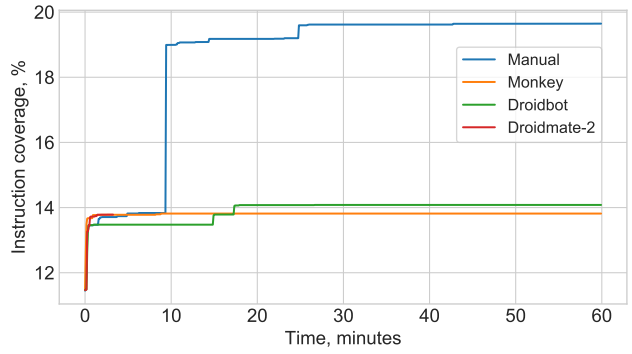

Fig. 1. Twitter Lite full instruction coverage under automated (Monkey, Droidbot, DroidMate-2) and manual explorations.

interacts with web page content. In turn, the web page calls appropriate methods in the app and Android APIs.

3) Exploration of behaviors: To explore Twitter behaviors, we used three automated testing tools, and we explored the app manually. Monkey [28], our first tool, is a popular, efficient and integrated into Android automated testing tool that randomly pushes UI events into an app. Second, Droidbot [27] is a more advanced tool based on Monkey but reinforced with the use of UIAutomator [36]. Droidbot recognizes particular UI elements and combines its actions based on this knowledge. In particular, Droidbot allows users to specify the login page id of the app and credentials to pass app authentication. The third tool is DroidMate-2 [26]. This state of art tool targets specifically 3rd-party apps. Like Droidbot, DroidMate-2 uses UIAutomator, but it provides additional test strategies and even allows anyone to develop a new. Last, we tested the app manually, which allowed us to run as much functionality as possible consciously.

4) Instruction coverage on observed behaviors: Figure 1 demonstrates the results of instruction coverage measurement when performing automated and manual exploratory procedures.

Our exploratory procedure worked as follows. Instrumented by ACVTool Twitter Lite app was installed on Android emulator (API 25), each automated testing tool worked for 1 hour as well as manual exploration performed by one of the authors. During this procedure, ACVTool generated instruction coverage every 5 seconds (in total, 720 values per hour).

We achieved the best results by manual exploration: 19.65% instruction coverage. Automated tools performed differently and demonstrated the following results: 14.08% Droidbot, 13.82% Monkey and 13.78% DroidMate-2. We ran the experiment twice and received very similar results. Among automated testing tools, Droidbot performed better, though it reached the peak slower. In the case of DroidMate-2, the tool managed to finish exploration in less than 3 minutes since it could not find more states (short red line appears in Figure 1).

Such a big difference in instruction coverage between the automated exploration and manual has its reasons. The main obstacle for automated tools was to sign up and log in. Though Droidbot and DroidMate-2 can to pass the login page in native apps (credentials need credentials need pre-configuration), the issue remains unresolved in the case of WebView apps, such as Twitter Lite.

Remarkably, both Droidbot and DroidMate-2 use UIAutomator under the hood. It helps tools to locate UI elements and extract their attributes. Therefore, Droidbot and DroidMate-2 are capable of recognizing login and password fields with specific identifiers. However, this is not the case in the WebView app. Here we confirm that the current implementation of UIAutomator is not able to pull field attributes out of a web page. Thus, both tools are not capable of passing either login or sign up pages in the WebView. This issue prevents automated tools from efficiently exploring such apps since a significant amount of functionality lies on the authorized side.

Indeed, during manual exploration, when passed the authorization, instruction coverage is significantly higher compared to automated tools. Coverage increased on more than 5% when tweeting a picture that we instantly shot with the camera, while sign up and login functionality took less than 0.3% of the codebase. Worth noting that after half an hour of exploration, we could not find more actions he could do to improve overall coverage. Therefore, we continued to interact with the app more randomly. However, *the peak coverage we managed to achieve is less than one-fifth of the number of instructions in the app*.

The Twitter Lite app experiment, in particular, demonstrated that neither absolute value nor relative increase of code coverage gave additional information on what or how to test more intensively. We could not also say for sure the maximum amount of runnable app code. Our experiment further raises questions. What does the rest of the app (more than 80% not executed)? Can we guarantee goodness of not executed code?

C. Interpretation

Many officials and even states leaders nowadays use Twitter. Potentially they may become a target to an attack through the app on their devices. An attacker may hide malicious functionality in the app that can trigger under specific conditions (logic bomb).

Indeed, Frantantonio et al. in their work [4] described logic bombs as a commonly employed by targeted malware mechanism that may be used in state-sponsored attacks. From the dynamic analysis perspective, we could not find the logic bomb. Moreover, on our samples, the percentage of instruction coverage did not give more understanding of time bomb behavior since coverage on *day 1* did not differ much from *day 0*. When measuring only the main package (code written by us), not covered code is present on both *days*.

Although we put in our Time Bomb app as little functionality as we could, while the Twitter app is quite complicated app, instruction coverage on both apps was under 20%. Furthermore, for the Time Bomb app, we used Proguard and aggressive R8 shrinking — modern tools that use static analysis to eliminate dead code and compress apps.

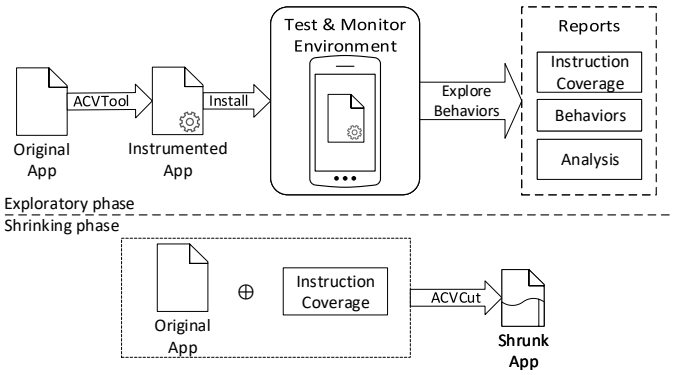

Fig. 2. Dynamic Binary Shrinking System for Android.

The experiments brought us to the conclusion that app instruction coverage depends on app functionality, including libraries and optimizations techniques. Hence, *the instruction coverage metric is not comparable between the apps*. In this case, testing techniques that rely on absolute code coverage values may not scale on different apps.

Moreover, the *maximum coverage value* for 3rd-party apps is always *unknown* due to the absence of requirements or a *complete test suite*. Code coverage, therefore, does not help to clarify when to stop testing too.

From the examples mentioned above, we can see that not only imperfection of automated tools contribute to the inability to reach 100% code coverage, nor dead code, but also the *extra code from libraries or written by app developers stays never executed*. This extra code constitutes the app's *bloat*, and thus, unnecessarily increases the attack surface [20].

IV. Methodology

Figure 2 demonstrates a high-level overview of the Dynamic Binary Shrinking System for Android. Using this approach, security specialists and test engineers may monitor an app for malicious activities while exploring its behaviors. When the exploratory procedure is finished, our tool, ACVCut, shrinks the app towards covered code. Thus, only executed code will remain in the shrunk version of the app.

The core of such a system is fine-grained code coverage. In this work, we use ACVTool [32], [33] — the only available fine-grained code coverage tool for Android. ACVTool provides code coverage information at the level of instructions, methods and classes. Full and precise information about executed instructions is essential for us since substantial modifications of a 3rd-party app may easily break it.

The Dynamic Shrinking System consists of two phases. The first phase aims to observe and analyze all relevant behaviors, while in the second phase, ACVCut shrinks the app. We further describe them in depth.

A. Exploratory Phase

During the exploration procedure, an analyst (or an automated tool) observes 3rd-party app functionality. The analyst may want to leave only specific behaviors or to test the app exhaustively. Thus, depending on the analyst requirements, the app may lose not essential features when shrunk. Therefore, in the best scenario, the app provider willing to pass checks with all behaviors may present a complete test suite for their app.

Software development companies with full development cycle maintain requirements documentation, unit-, integration and regression tests, mock objects, and manual test cases with a simple goal — to check all required features. We consider these testing artifacts as a complete test suite.

This approach is beneficial to both sides since the app provider is interested in delivering a well tested app. At the same time, the analyst needs to make sure that the app does not perform malicious actions. Therefore, an app provider could shrink the app himself and deliver it together with the complete test suite, so that anyone could verify the app on 100% coverage.

The exploratory phase from a technical perspective goes as follows. The original APK gets instrumented by ACVTool to allow code coverage measurements. Then we install the instrumented app onto a device or emulator. An analyst may enable dynamic analysis tools to monitor which sensitive actions the app performs. Thus, by the end of testing, we may obtain three outputs: instruction coverage, explored behaviors and app activity analysis reports.

From the obtained reports, instruction coverage is generated by ACVTool and goes as an input to the Shrinking phase. The behaviors report represents a complete test suite to the next generated shrunk version of the app. The analysis report aims to confirm the absence of malicious activities that happened under the exploratory phase. The analysis report relies upon *Monitor Environment* used to track suspicious activities.

This work aims to present a shrinking approach and its applications, which brings Dynamic Binary Shrinking System to life. Therefore, the monitor environment is not the goal of this paper. However, monitoring is an essential part of the system to confirm that the shrunk app is benign. We suggest using state-of-the-art approaches to monitor sensitive APIs and HTTP requests. Further, the shrunk app loses a significant part of the code. Therefore, existing static analysis approaches may benefit from significantly smaller codebase available for analysis.

B. Shrinking Phase

The instruction coverage report specifies the exact code executed during the testing phase. Therefore, we can remove all not tested functionality. In a nutshell, we match the original `smali` code on our code coverage report and, through dedicated code manipulations, remove all unnecessary functionality. Such operation is not trivial and requires attention to specific Android bytecode architecture since we can easily break the app. We give details on the `smali` code modification in Section V.

C. ACVCut implementation

ACVCut heavily relies on ACVTool under the hood, which performs a full cycle of app repackaging. ACVTool instru-

ments the smali code, initiates the instrumentation process before the beginning of the exploratory procedure, saves runtime report at the end of exploration and finally generates code coverage report [32], [33]. Here we give more details on the design of ACVCut.

ACVCut starts form the preparation of the working directory with the help of ACVTool. The directory should include the instrumented by ACVTool app, instrumentation report and the decompiled app directory. The instrumented app takes its part in the exploratory procedure and produces the runtime report. Runtime reports contain instruction coverage information in the form of binary arrays and, when applied to instrumentation reports, produce tree-based smali code structure, where executed instructions are marked as covered. ACVCut walks through the tree in several passes to determine and remove not-executed basic blocks and remove or make *stubs* from not called methods. *Stub* methods removal may break the app due to its structural role, e.g. they may take part in the inheritance hierarchy. However, *stubs* have no executable code and are never called. Therefore, they should be excluded from all types of analysis. When code processing is finished, ACVCut builds the shrunk app with the help of apktool. Finally, the output of ACVCut contains the shrunk app and a list of neutralized *stub* methods.

V. Cutting The Code

In this section, we describe our approach to removing the not-executed smali code from an app. The approach performs an extensive smali code processing and a lightweight static analysis for solving challenges related to maintaining consistency, coherence, and cohesion of complex compiler-generated code. We first get rid only of instructions in executed methods and further remove not executed methods.

A. Instructions

In this work, we rely on the smali representation since it gives us the possibility to modify the app binary at the finest granularity.

1) Basic Block: Besides the fact that ACVTool currently supports instruction- and method-level code coverage, we found it is more efficient to operate on smali basic blocks. We made this technical decision because ACVTool does not mark specific instructions when it is impractical or impossible to insert probes [33].

We focused on basic blocks since, normally, instructions in a basic block either executed all or none. This strategy works well with the only special case when one instruction throws an exception. Then, the instructions next to the failing instruction stay not executed. This case is easy to find by checking if the last instruction of the basic block worked. Further we describe our study on basic block definition for the smali representation.

Basic block contains a sequential branch-free set of instructions that starts with a labeled operation and ends with a brunch, jump or predicated operation [37]. In the smali representation, we developed the following rules. A basic block may start with the first instruction in the method, a label or the if-* instruction. The basic block ends on the last instruction before the beginning of the next basic block, the goto instruction or the end of the method.

ACVCut gets rid of all not executed basic blocks. Removal of a basic block means sequential removal of instructions from it and removal of corresponding labels. In our implementation, we extended the definition of a basic block to a *labeled block* since all instructions between two labels are always covered or none of them. However, basic block removal is still a challenging task since executed branch instructions (if-* or goto) may stay in place and reference to the deleted basic block label.

For instance, statement "if-eqz v7, :cond_0" compares the v7 register value to zero. When the condition was satisfied, a program pointer jumps to the label :cond_0. Otherwise, the program pointer passes the statement to the next instruction. ACVCut may remove the basic block corresponding to the specified label (:cond_0) when they were not executed. In this case, the "if-eqz v7, :cond_0" statement becomes invalid since the target label declaration is absent now. To solve this issue, we analyzed possible cases and we suggest code manipulations that eventually modify the original program control flow.

2) Conditional jump logic: The issue of an absent label reference, mentioned above, may occur to all instructions able to reference a label: if, goto and .catch. Since goto always finalizes the basic block and gets removed along to the not executed basic block, it does not lead to the issue in our implementation. However, if and .catch statements need a more sophisticated approach. We describe a solution to the if statement that references an absent label and further explain the .catch case in the next *Try-Catch* paragraphs.

The if instruction in the smali representation has the same meaning as in the Java code. It has four combinations depending on the coverage of the if instruction itself and the coverage of the basic block referenced by the specified label. Concerning the case, we do the following.

We check each targeted label specified for the executed if statement. If the label does not appear in the code anymore, ACVCut removes the if statement since it did not perform conditional jump during the test.

The opposite case is when ACVTool did not mark the if statement. It does not always mean that the statement did not execute. Indeed, the statement could work and make a jump to the specified label. Therefore, the probe placed next to the statement did not run, and ACVTool could not mark the appropriate statement. In this case, we check if the previous instruction was run. This means to us that the program pointer reached if statement and performed a conditional jump to the specified label. Since only one branch worked, we replace the statement and the rest of the basic block with the unconditional goto.

3) Try-Catch: The exception mechanism in the smali representation relates to the Java try-catch structure. The monitored block starts and ends with :try_start and

:try_end labels correspondingly, and it is followed by the .catch and .catch-all meta-instructions. They first declare the type of the exception try-catch block catches, second — where the monitored code starts and ends (labels :try_start and :try_end), and third — the target label where to handle the exception.

When the try-catch is covered, we check exception handlers mentioned in the .catch meta-instruction. If the code did not throw the specified exception, then the corresponding handling basic block did not work too, and was removed in previous steps. Therefore, we remove the .catch as well. If no .catch meta-instructions are left in try-catch, we remove :try_start and :try_end labels too. Thus, if the code did not throw an exception, it would continue working smoothly. The opposite, when the code throws an exception, the try-catch and handling code stay in place. Moreover, an instruction that threw an exception is not marked as covered (since the probe next to it was not executed), but it has to stay in code and perform its function to throw the exception.

4) Synchronized code: Complex apps may deal with multiple threads that often need access to a shared resource. In this case, developers use the Java keyword synchronized to allow one thread at a time to the selected code. Android implements this mechanism using a monitor object and a pair of instructions monitor-enter and monitor-exit to specify the beginning and the end of synchronized code.

Synchronization is sensitive to exceptions and can affect the work of a device. Hence, an app is required to unconditionally catch possible exceptions in synchronized code and call emergency monitor-exit (contains two instructions). Android compiler takes responsibility for developers to generate the required exception handler with additional monitor-exit code, so that the app code could pass the Android Runtime Verifier checks concerning the synchronized code.

In the same way, we need to follow the Android Runtime Verifier rules when modifying the smali code. When synchronized code worked without any exception, the generated try-catch can be removed. However, we leave the monitor-exit code in place since it helps the app to pass the Runtime Verifier. As a result, monitor-exit code stays as an exceptional addition that may never be called.

5) Switches: The switch statement in Java represents a set of cases that may be chosen depending on the passed value. The Android Compiler translates the Java switch statement into either the packed-switch or the sparse-switch instruction, depending on the value sparseness in the cases.

The switch instruction consists of two parts in smali . The first part declares the switch statement in the code, while the second part stays at the end of the method and contains a tuple of cases. Each case is a pseudo-instruction that references a label attached to the basic block to handle a specific case. The whole second part cannot be tracked with ACVTool. Therefore, we again take a close look at which labels stay in the code.

Since we removed not executed basic blocks, we also need to get rid of the corresponding pseudo-instructions that reference absent labels. However, the removal of a pseudo-instruction violates the original switch table value allocation. Instead of removal of a case, we replace it with another working case in the original order. Though a switch statement may have a series of identical pseudo-instructions, its logic remains correct for the remaining cases. If no cases worked during the test, we remove both parts of the switch instructions.

6) Arrays: Similarly to the switch statements, arrays statements also have two parts, where the first part declares an array (the fill-array-data instruction), and the second part keeps enumerated array values. When the fill-array-data instructions were cut out from the code, the corresponding .array meta-instruction containing array values gets removed too.

7) Merging gotos: As a result of our above-mentioned code manipulations, we noticed many cases when a goto instruction jumps right to the next line. This operation is equal to standard program pointer increment and happens due to the removal of basic blocks between the goto instruction and the label it jumps to. We remove such a goto instruction and the label if it has no other references.

B. Methods

Above, we described a sophisticated analysis of smali structures on instruction removal. We perform this approach only on called methods since other methods stayed untouched under exploration and, intuitively, should be deleted. However, the removal of not used methods is not an obvious task too.

1) Methods removal: Android picks up object-oriented features of Java, such as *encapsulation*, *polymorphism* and *inheritance* for its bytecode. Therefore, Android also has notions of abstract methods and classes, interfaces and virtual tables for inheritance. When calling the methods, Android uses the corresponding invoke-* instructions.

Even when not called, a method may participate in the inheritance hierarchy or polymorphism taking place in the corresponding virtual table at run time. Such a method is usually called through the invoke-super or invoke-virtual instructions that take into account the object type hierarchy at run time. Predicting the correct run time type based on the code coverage and class hierarchy and altering method calls accordingly is not a part of this study. We see it as an exciting field of research from the perspective of bytecode optimization. However, this issue does not relate to *static* and *direct* methods, and we can remove them for sure.

2) Stub methods: The rest of the methods will stay in the app since some of them maintain an object-oriented skeleton of the app, as we have mentioned earlier. However, here we do the following trick. We first remove all the instructions inside the not called method and the second — put the default return value according to the declared type. One particular case here is the constructor of an inherited class. We add a statement to call the default Ljava/lang/Object;-><init>V() constructor, since the inherited constructor always calls the constructor the superclass.

Thus, the app has many empty methods that we call stub methods. ACVCut saves signatures of such methods into a list so that other tools could exclude stubs from the analysis.

C. Offensive mode

When removed all extra instructions, some not explored functionality could disappear. Though we consider not tested code to be potentially malicious, the absence of legitimate features can be unexpected to the end-user. For instance, the user may press a not tested button. Then, the button will not react since now its method-listener is empty.

A friendly approach to address this issue is to bring into the app an exception code to alert the end-user that the functionality was purposefully cut.

VI. Results

In this section, we applied our shrinking technique to the running examples.

A. Shrunk Time Bomb

We shrunk the app on *day 0* to verify what happened with the time bomb behavior on *day 1*. To evaluate the results of app shrinking, we also instrumented the shrunk version with ACVTool and measured the instruction coverage achieved by the complete test suite described in Section III-A1.

Table I demonstrates the results on shrinking Time Bomb on *day 0*. When testing the app, we observed the same behavior, as we described in Section III. Indeed, instruction coverage of the shrunk version shows 99.9% since we removed all the extra code. Additional benefits are smaller app size and less code. The decompressed binary DEX file decreased almost twice, while the amount of instructions, methods and classes greatly decreased in 9, 7 and 4 times correspondingly. App size changed slightly because the DEX file is saved in a compressed form, while app resources took the most of space.

TABLE I
THE ORIGINAL AND THE SHRUNK TIME BOMB APP METRICS ON DAY 0

App Version	Coverage	Instructions	Methods	Classes	APK Size	DEX Size
Original	15.2%	58045	3002	388	784 KB	591 KB
Shrunk	99.9%	6342	431	102	701 KB	327 KB
Profit	-	x9	x7	x4	12%	x2

The app behavior remained unchanged on *day 1*. The time bomb functionality disappeared. However, there are cases when apps have similar benign functionality. If we wanted to keep both behaviors, we have two options. First is to test both behaviors separately and merge their instruction coverage before shrinking the app. Second, specific functions can be shortlisted to exclude their shrinking.

a) Not covered instructions.: It is worth noting why instruction coverage is very close to 100% but not precisely. ACVTool marked only 7 instructions as not covered. Four of them are necessary to respect the Runtime Verifier rules concerning the synchronized code. Three others relate to the handled exception. We describe the reasons in Section V-A.

B. Shrunk Twitter Lite App

On the Twitter Lite app, we performed the following experiment. We took instruction coverage generated by Monkey as an example of automatically explored behaviors keeping in mind that it did not pass the login page and did not see a significant part of the app. Since this is a WebView app, we expect that shrinking preserved the major part of functionality except for not observed functions that closely work with Android APIs. The goal of this experiment is to determine what behaviors continue to stay in the app and how the user interface (UI) responds to shrinking.

Table II presents changes that happened to the app after shrinking. Though the size of the app was already small compared to the full Twitter version, the app size decreased to 1036KB from 1327KB after shrinking. The number of instructions, methods and classes fell dramatically: 14, 7 and 3.5 times less than in the original app.

TABLE II
THE ORIGINAL AND THE SHRUNK TWITTER LITE APP METRICS

App Version	Coverage	Instructions	Methods	Classes	APK Size	DEX Size
Original	13.82%	166031	11140	1680	1327 KB	1896 KB
Shrunk	97.9%	11198	1562	488	1036 KB	1757 KB
Profit	-	x14	x7	x3.5	28%	≈

Remarkably, the app preserved all web functionality that Monkey could not observe, such as logging in, tweeting, interacting with other profiles (follow/unfollow, like, comment, retweet), sending and receiving messages, profile editing, changing theme color, changing app language and other app settings. However, other not observed features, that rely on interacting with the Android code, disappeared. Indeed, the app lost notifications, sharing a tweet to other apps, sharing pictures and text from other apps to Twitter, posting pictures, changing user avatar and the background picture. Moreover, when trying to tweet a picture, we tapped on the *attach picture* button, the button animation worked. However, the standard dialog did not pop out to choose between the gallery and camera pictures. The same happened when we tried to change the user avatar and background picture.

Thus, when exploring behaviors before shrinking the app, we can decide on what functionality we want to leave. This particular ability allows us to impose privacy requirements on the app. For instance, when an organization tries to regulate the use of apps on employees' devices, it may allow them to access social networks but restrict posting sensitive pictures. Moreover, the app will never ask the user for a suspicious permission if the corresponding functionality was deleted. Thus, the user of the shrunk app has no chance to give a permission by mistake. For instance, our shrunk Twitter Lite app preserved most of the major features while restricted access to sensitive Android APIs.

VII. Discussion

In this section, we shortly review existing threats and limitations that we may address in future work.

A. Threats and limitations

Internal validity. Threats to internal validity relate to the implementation of our tool and discussed methodology on Dynamic Binary Shrinking System.

Although our tool processed a significant amount of obfuscated code contained in our samples, the tool may have bugs and may break some apps. We will continue the development of ACVCut and evaluate it on more apps.

The stub methods continue staying in the code. However, we save them into a list of stubs to exclude from other analyses.

Our approach may shrink legitimate, however, not tested behaviors. Only app reviewer can decide which functionality to leave in the app. However, interested app producers can share their full tests addressing all the app requirements.

Android applications framework contains backward compatibility code that we shrink too. The app may be tested and shrunk for the device model, where it is expected to work.

External validity. We acknowledge the following threat to the generalization of our findings. A simple time bomb example does not scale on the possible diversity of logic bombs and attacks. Although we decreased the attack surface, an attacker may maliciously use existing app vulnerabilities to exploit benign app capabilities.

ACVCut limitations. Since our tool depends on the ACVTool, it inherits all the limitations of ACVTool. It is impossible to apply our approach when ACVTool is not able to generate instruction coverage. We refer to the original ACVTool paper for the limitations [33].

B. On the ACVCut development

The development process of ACVCut is far from trivial. Unfortunately, `smali` language suffers from a lack of documentation and the absence of code practices commonly available for ordinary programming languages. The language itself is pretty low-level; it is an assembler. No wonder its learning curve is pretty steep. These limitations badly contribute to ACVCut development because it requires a rare skill-set that would allow debugging potential errors injected by ACVCut into a 3rd-party app.

Doing this work, we transformed many observed `smali` code cases and rules (see Section V-A) into an automated tool. Taking into account the flaws mentioned earlier, the development of the tool became more evolutionary. Although ACVCut worked well on the running examples (Section VI), it may break some other apps. We will continue to generalize our code processing approach along with shrinking more apps.

We also improved the correctness of instruction coverage generated by ACVTool. Notably, we fixed coverage calculating for `.catch` meta-instructions (incorrect coverage may inject errors into ACVCut). We also fully reworked the Instrumentation class to allow on-demand coverage reporting at any time. Moreover, ACVTool can now track instructions even on the app termination since we keep the instrumentation process running all the time. These updates are essential for our tool because apps are sensitive to careless code manipulations and break easily. When coverage information, for instance, is absent on app termination, the shrunk app would be crashing on each app exit. We will pull request these updates into the original ACVTool repository after the acceptance of this work.

VIII. Related work

Program debloating is in the closest relation to our approach since it describes program thinning and the removal of specific features. It found applications in such areas as debloating of libraries, source code, containers, hardware, and in delta debugging. Debloating techniques emerged in several recent works and available now for various platforms to reduce programs with and without source code.

Brown et al. remove features from software based on source code to feature mapping [38]. Azad et al. remove complete features in web applications by using a PHP profiler to extract code coverage information [21]. Heo et al. developed the Chisel system to effectively customize C programs with the help of reinforcement learning [19].

Agadakos et al., in their recent work, developed Nibbler, a tool for bloat removal from binary shared libraries. Nibbler works on x86 binaries, does not recompile the program and relies on static analysis to eliminate not reachable code [20]. Landsborough et al. make software thinner with two approaches: first relies on dynamic tracing as a guide while the second uses a genetic algorithm to mutate a program [39]. The authors also argue on the pros and cons of thinned programs from security, size, validity and optimality perspectives.

In Android, researchers rely on static analysis to remove dead code. Jiang et al. proposed JRed and RedDroid approaches to remove bloat from Java applications, Java Runtime and Android apps [16]–[18]. Proguard, a Gradle plugin for shrinking Android apps, performs static analysis to cut dead code [30].

While static analysis and debloating are well researched, to the best of our knowledge, Android did not have a solution for thinning apps based on the app execution traces or code coverage.

IX. Conclusions

In this work, we presented Dynamic Binary Shrinking System — a novel methodology created for shrinking 3rd-party Android apps towards observed benign behaviors. We developed a sophisticated technique on modifying 3rd-party apps and incorporated it into an open-source ACVCut tool.

Our running examples we demonstrated several findings. First, large amounts of not used code stay in apps. Second, the measured coverage does not provide information on whether all legitimate app behaviors have been seen.

Dynamic Binary Shrinking System can guarantee the absence of potentially malicious code with 100% instruction coverage. We shrunk two apps and confirmed the viability of our approach. However, shrinking may lead to the disappearing of legitimate features if they stay not tested.

Acknowledgements

We would like to thank Prof. Dr. Sjouke Mauw for valuable feedback. This work is supported by the Luxembourg National Research Fund (FNR) AFR-PhD-11289380-DroidMod.

References

[1] A. Pilgun, "[GitHub] pilgun/acvcut: ACVCut v1.0, Zenodo," Sep 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4060422

[2] Google. (2020) Google play protect. [Online]. Available: https://www.android.com/play-protect/

[3] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 1–41, 2017.

[4] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 377–396.

[5] Google. (2020) Mobile device management system for android enterprise. [Online]. Available: https://www.android.com/enterprise/management/

[6] M. Linares-Vásquez, G. Bavota, and C. Escobar-Velásquez, "An empirical study on android-related vulnerabilities," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 2–13.

[7] S. Dashevskyi, Y. Zhauniarovich, O. Gadyatskaya, A. Pilgun, and H. Ouhssain, "Dissecting android cryptocurrency miners," in *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, 2020, pp. 191–202.

[8] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.

[9] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, 2018.

[10] K. Jamrozik and A. Zeller, "Droidmate: a robust and extensible test generator for android," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, 2016, pp. 293–294.

[11] L. Bao, T.-D. B. Le, and D. Lo, "Mining sandboxes: Are we there yet?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 445–455.

[12] T.-D. B. Le, L. Bao, D. Lo, D. Gao, and L. Li, "Towards mining comprehensive android sandboxes," in *2018 23rd International conference on engineering of complex computer systems (ICECCS)*. IEEE, 2018, pp. 51–60.

[13] Y. Zhauniarovich and O. Gadyatskaya, "Small changes, big changes: An updated view on the android permission system," in *Research in Attacks, Intrusions, and Defenses*, F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, Eds. Cham: Springer International Publishing, 2016, pp. 346–367.

[14] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Automatically securing permission-based software by reducing the attack surface: An application to android," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2012, pp. 274–277.

[15] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 37–48.

[16] Y. Jiang, C. Zhang, D. Wu, and P. Liu, "Feature-based software customization: Preliminary analysis, formalization, and methods," in *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 2016, pp. 122–131.

[17] Y. Jiang, D. Wu, and P. Liu, "Jred: Program customization and bloatware mitigation based on static analysis," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, 2016, pp. 12–21.

[18] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu, "Reddroid: Android application redundancy customization based on static analysis," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 189–199.

[19] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 380–394. [Online]. Available: https://doi.org/10.1145/3243734.3243838

[20] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating binary shared libraries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 70–83. [Online]. Available: https://doi.org/10.1145/3359789.3359823

[21] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: quantifying the security benefits of debloating web applications," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1697–1714.

[22] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "{RAZOR}: A framework for post-deployment software debloating," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1733–1750.

[23] L. Onwuzurike, M. Almeida, E. Mariconti, J. Blackburn, G. Stringhini, and E. De Cristofaro, "A family of droids-android malware detection via behavioral modeling: Static vs dynamic analysis," in *2018 16th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2018, pp. 1–10.

[24] 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*, ser. CASCON '10. USA: IBM Corp., 2010, pp. 214–224. [Online]. Available: https://doi.org/10.1145/1925805.1925818

[25] Google. (2018) smali. [Online]. Available: https://android.googlesource.com/platform/external/smali/

[26] N. P. Borges Jr, J. Hotzkow, and A. Zeller, "Droidmate-2: a platform for android test generation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 916–919.

[27] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017, pp. 23–26.

[28] Google. (2020) UI/Application Exerciser Monkey. [Online]. Available: https://developer.android.com/studio/test/monkey

[29] J. Rabe. (2016) Timebomb: Stops app usage after a period of time has passed starting from app build date. [Online]. Available: https://github.com/kibotu/TimeBomb

[30] GuardSquare. (2020) Proguard manual. [Online]. Available: https://www.guardsquare.com/en/proguard/manual/introduction

[31] Google. (2020) Shrink, obfuscate, and optimize your app. [Online]. Available: https://developer.android.com/studio/build/shrink-code

[32] A. Pilgun, O. Gadyatskaya, S. Dashevskyi, Y. Zhauniarovich, and A. Kushniarou, "An effective android code coverage tool," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2189–2191.

[33] A. Pilgun, O. Gadyatskaya, Y. Zhauniarovich, S. Dashevskyi, A. Kushniarou, and S. Mauw, "Fine-grained code coverage measurement in automated black-box android testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–35, 2020.

[34] I. Google. (2020) WebView: A View that displays web pages. [Online]. Available: https://developer.android.com/reference/android/webkit/WebView

[35] T. O. Foundation. (2020) Mobile security testing guide. android anti-reversing defenses. [Online]. Available: https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05j-testing-resiliency-against-reverse-engineering

[36] Google. (2019) UI Automator. [Online]. Available: https://developer.android.com/training/testing/ui-automator

[37] K. Cooper and L. Torczon, *Engineering a compiler*. Elsevier, 2011.

[38] M. D. Brown and S. Pande, "Carve: Practical security-focused software debloating using simple feature set mappings," in *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, 2019, pp. 1–7.

[39] J. Landsborough, S. Harding, and S. Fugate, "Removing the kitchen sink from software," in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 2015, pp. 833–838.