Dynamically Blocking Target Methods in Android Applications

Zicheng Zhang
Singapore Management University
Singapore, Singapore
zczhang.2020@phdcs.smu.edu.sg

Jiakun Liu
Singapore Management University
Singapore, Singapore
jkliu@smu.edu.sg

Debin Gao
Singapore Management University
Singapore, Singapore
dbgao@smu.edu.sg

David Lo
Singapore Management University
Singapore, Singapore
davidlo@smu.edu.sg

1 INTRODUCTION

In current era, mobile applications (apps) have become an import part of our daily lives, which has significantly changed our lifestyles. However, developers tend to add more and more functionalities in the app, while most of them are used infrequently. Those unused code may take up the memory space of the device and affect the performance while user are using the apps. Moreover, the unused code poses a security risk, as it can be exploited for code re-use attacks, e.g., Return Oriented Programming (ROP) attacks. What's worse is that if the unused code contains malicious code and they could be exploited by adversaries even if users do not use them, . To address these issues, we propose an approach for dynamically blocking a list of target methods while running Android apps.

2 BACKGROUND

Android Runtime (ART). Android runtime is a new mechanism introduced in Android 4.4, and it became the default runtime environment since Android 5.0, which uses ahead-of-time (AOT) compilation that compiles all the DEX code into native instructions during the app installation. Starting from Android 7.0 until now (Android 13), ART uses a hybrid combination of AOT, just-in-time (JIT), and profile-guided compilation. Based on this mechanism, an app is initially installed without AOT compilation, but a new file called OAT file (in an extended ELF format) is generated with only its Dalvik bytecode (also called DEX code). Every time a method is invoked, the ART sends the method's DEX code to an interpreter, which interprets the DEX code into ARM instructions and executes them. When a method is frequently invoked, it will be JIT compiled into native code and stored in a code cache, and the method will be added to a profile. When the device is idle and charging, a compilation daemon AOT-compiles those frequently used code to native instructions based on the generated profile, and inserts it into the OAT file. Generally, each app has its own runtime instance, which is established while launching the app and preserves all the running configuration and status of the app. Both the app and their ART are running in the same process with the same Linux user ID (uid). Since the permissions are granted based on the uid, the runtime instance shares the same permissions as its corresponding app.

Native library loading. Android allows applications to incorporate native libraries, which are mostly *so* libraries, and use JNI to invoke native methods from the Java side. Since Android 6.0, the Android system supports loading uncompressed so libraries from the APK file. The app can load a specific so library from the APK by using the API `System.loadLibrary()`, after which they can invoke the native method within the library. The so library is an extended ELF format file, which contains several headers and their corresponding sections. When loading the *so* library, the Android system first reads the headers to locate the start and end of the contents to be loaded in the memory and allocates ample enough space in the memory, after which it starts to load the actual content into the memory by mapping different sections in the file into different segments in the memory through a system call `mmap()`.

3 PROPOSED APPROACH

3.1 Assumption

The primary purpose of our framework is two-fold: first, it aims to intersect the execution of specific methods, and second, it defends against code reuse attacks such as potential Return-Oriented Programming (ROP) attacks. Remarkably, this functionality is achieved without static modifications to the APK file. This report assumes that the schema, which comprises a list of methods to be removed, has already been obtained. This schema can be generated using existing static debloating tools [17, 22], provided by anti-virus engines or defined manually by the user. Users have the flexibility to

modify this schema through a management application. Our framework is integrated into a customized and unrooted Android OS on the user's device, instrumenting the Android runtime (ART) [3], Android framework [2] and Bionic module [4]., while leaving the kernel unmodified. Users can use the phone independently, i.e., without connecting to the computer. During installation, applications do not utilize a profile to indicate which parts should be AOT-compiled. Consequently, all DEX methods remain uncompiled during installation. Only native code (i.e., ARM instructions on Android devices) can be used as the ROP gadgets. Apps may attempt to detect our framework through side channels and potentially evade blocking by force-stopping their apps.

3.2 Overview

The overall workflow of our framework is illustrated in Figure 1. The process can be divided into three parts: sharing blocking schema, blocking DEX methods, and blocking native library methods. Firstly, to conduct method-blocking, our framework needs a list of methods to be blocked, i.e., the blocking schema. The schema is managed by a management app and shared with other apps and their Android runtime environment (see ① in Figure 1) through ContentProvider, a data sharing mechanism. Meanwhile, to successfully obtain the blocking schema, we must grant read permission to every third-party app. The details of the first part are elaborated in § 3.3. Then, each time an app is launched, the instrumented ART reads the schema via the ContentProvider, after which our framework can block methods during the running of the application. The blocked methods include DEX methods and native library methods. To block the DEX methods, Our framework utilizes the hybrid compilation mechanism of the Android system (see ② in Figure 1). It intercepts the method invocation and prevents it from being passed to the interpreter. Additionally, our framework freezes the method counter of target methods to prevent these methods from being JIT/AOT-compiled after they are frequently invoked. Details are described in § **??**. For methods in native libraries, our framework blocks them by instrumenting the process of both library loading and method invocation (see ③ in Figure 1). While loading the native library, our framework locates the offset of the target method in the library and calculates its actual memory address, after which it zero-fills the target space in memory to prevent loading blocked methods into the memory. Then, when the blocked method is invoked, the entry point of that method will be linked to the start address of an empty memory space. Details are described in § 3.5. Moreover, to provide a user-friendly interface for the DEX and native method blocking, our framework provides a graceful termination when a blocked method is invoked. The instrumented ART will start an Activity of the management app to display necessary information.

3.3 Sharing blocking Schema

Before dynamically blocking target methods, we must establish an inter-procedural communication channel between our management app and the Android runtime. This channel will facilitate sharing the blocking schema, allowing the runtime to identify which methods should undergo blocking. It is worth noting that the Android runtime is implemented in C/C++, providing the execution environment for Android apps primarily written in Java/Kotlin. Therefore, we need a cross-language approach to share the schema successfully. However, we cannot store this file within the app's public data folder, as doing so would require other apps to request `READ_EXTERNAL_STORAGE` permission to read the schema, since the runtime shares the same permissions as the app. In the current state of research, many approaches rely on a configuration file to exchange information between the app and ART (Android Runtime). For instance, FA^3 [15] utilizes a configuration file placed in a publicly accessible folder such as `/data/local/tmp`. This allows runtime instances to access and read the configuration file during the app launch. However, a significant drawback of this method is that malicious apps could potentially manipulate the file, removing their methods from the blocking schema. As a result, their malicious functions could continue to operate unaffected during the blocking process.

To address the vulnerability of malicious apps modifying the blocking schema, we propose a new communication channel that ensures only the management app can modify the schema. In contrast, all other apps can only read it. We achieve this by leveraging Android's ContentProvider [5] and Java Native Interface (JNI) [6].

Here is how our approach works:

- Management App Setup: The management app creates a dedicated database to store the blocking schema. Additionally, it establishes a ContentProvider that offers interfaces to manage access to the database. Users can then use the management app to configure the blocking schema.
- Read-Only Permission: The management app defines read permission for the ContentProvider, which permits other apps and their runtime environment to read the content in the database but restricts them from making modifications.
- Sharing Schema with ART: When an app is launched, our framework employs JNI to invoke the read interface of the ContentProvider from ART (written in C/C++), allowing it to load the blocking schema into

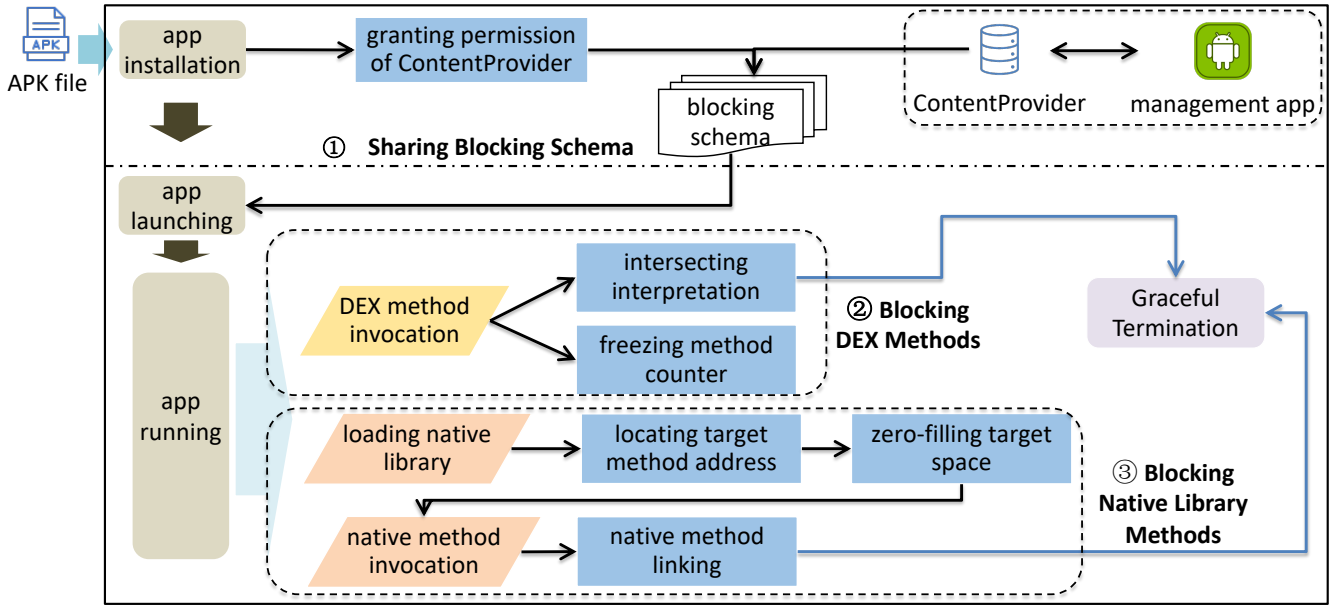

Figure 1: The overall workflow.

the runtime instance. This process completes the schema sharing from the app to the ART.

- Context Wait: Since the interfaces of the ContentProvider require the context of the app, we need to wait until the app's context is created after its launch before proceeding with schema sharing.
- Permission Grant Modification: To simplify the process for third-party apps and avoid requiring them to explicitly declare the read permission for our ContentProvider in their manifest files, we modify the permission grant process of the Android frameworks. We automatically insert the read permission into the requested permission list of every third-party app. Importantly, this permission is categorized as normal permission, and the system can automatically grant it after the app installation.
- Blocking Schema Update: The process is straightforward when users decide to modify the blocking schema. They simply need to make the necessary changes through the management app and restart the target application. Upon restarting the target application, the ART automatically loads the updated blocking schema via the ContentProvider.

By implementing this communication channel, we balance enabling legitimate apps to access the blocking schema and preventing malicious apps from making unauthorized changes.

It is crucial to note that our approach does not involve any static modification to the APK file. Instead, we dynamically insert the permission after installing the app. Since the inserted permission is a read-only permission and the content within the database is entirely under the management app's control, there are no potential risks to user security and privacy. This design ensures that only the management app maintains full authority over the blocking schema, minimizing any potential harm from unauthorized modification.

3.4 Blocking DEX Code Methods

It is important to note that the DEX code in apps cannot be directly executed; instead, it needs to be passed to the interpreter for further compilation. In the Android Runtime (ART), each method within an app corresponds to an ArtMethod object, which preserves essential information such as the method name, code address in the DEX code, entry point, and method counters. When a method is invoked, ART checks if the method has been loaded before. If not, it loads the method from the app's DEX code and sets the entry point to a stub called `art_quick_to_interpreter_bridge`. This stub checks the DEX code and passes it to the interpreter, which interprets the code into native instructions for execution. Additionally, each method maintains a method counter that records the number of times the method has been invoked. If the counter exceeds a threshold, the method is JIT-compiled or AOT-compiled into native instructions,

and the method's entry point is changed accordingly. Regarding DEX method blocking, we focus not on removing the DEX code from the app entirely but rather on preventing it from being compiled into native code. We need to address two critical problems: 1) interception of DEX Methods: We must intercept the DEX methods that are being interpreted into native code and execute only those that do not belong to the blocking schema. For methods in the schema, we prevent their invocation and return null, making it appear as if the method body is empty; 2) Prevention of JIT/AOT Compilation: To achieve DEX code blocking, we must ensure that the DEX code is not JIT-compiled or AOT-compiled into native code.

Specifically, we modify the interpreter to check if a method belongs to the blocking schema before it is interpreted into native code. If so, it intercepts the method invocation and returns null as if the method body is empty. Meanwhile, our framework resets the method counter of the target method in its ArtMethod object so that it will not trigger the JIT/AOT complication. For methods outside the schema, their DEX code is interpreted and executed as usual. Furthermore, to provide a graceful termination and inform the user that the system has blocked the method rather than encountering an app execution error, our framework utilizes JNI to start an Activity of the management app. This Activity then displays the necessary information to the user. By implementing these modifications, our framework effectively achieves DEX method blocking by selectively intercepting method invocations and preventing the compilation of DEX code into native instructions, thereby conducting DEX method blocking while ensuring user awareness of the blocking process.

3.5 Blocking Native Library Functions

In addition to DEX code, app developers widely use native libraries for their ability to make system calls. However, these libraries are also susceptible to facilitating malicious behaviors, as highlighted in prior research [9, 19, 21]. Unlike DEX methods, native library methods can be directly executed, rendering the previously mentioned blocking approach inapplicable. More than simply intercepting method invocations is required. As long as the native methods are loaded into memory, they can be exploited as gadgets for code reuse attacks. Therefore, the focus should be on preventing loading native methods listed in the blocking schema into memory. In other words, we need to selectively load the methods, excluding those specified in the schema. However, this presents two non-trivial challenges. Firstly, when loading the shared object library (*so* library), the lazy linking mechanism delays linking ArtMethod objects to their corresponding method code in the loaded memory until the method is called. Since our blocking schema relies on method descriptions, typically their names, the first challenge is to locate the offset of target methods in the *so* library based on their method names. Secondly, the system call `mmap()` necessitates the starting offset loaded in the file to be an integer multiple of the memory page size. Consequently, it is impossible to load each method individually unless we modify the kernel and alter the system call implementation, as multiple native methods may exist within the same memory page. How to selectively load the native methods without kernel modification is another challenge.

To overcome these challenges, our framework employs a strategic approach that involves linking the names of blocked methods with their method code by reading extra headers and removing the blocked method code from memory through zero-filling. Specifically, during the process of reading the section headers of the *so* library, our framework reads and stores two additional headers called `dynsym` and `dynstr` from the so file, which are not initially loaded until the ArtMethod linking process takes place. The `dynsym` serves as a symbol table, preserving the starting offset, size, and an index for identifying each native method. On the other hand, the `dynstr` contains a list of all symbol names in the library. By utilizing the index from `dynsym`, our framework can determine the actual method name in the `dynstr`, thereby obtaining the starting and ending offset of the target methods in the so file. The address of the target methods in the memory can be calculated by combining the page and method offsets.

Having obtained the necessary information, our framework addresses the second challenge while mapping library sections into memory segments. It checks if the mapped content contains the target method code, accomplished by comparing the mapped offset of the content with the address of the target methods. If a match is found, our framework employs a specific procedure to ensure selective loading and protection of native methods. our framework performs zero-filling on the mapped memory space of each target method based on the obtained address and size. After that, our framework insert an eight-byte native code which contains instructions to return null as if the function body is empty. This step is to make the app running more robust, otherwise the app will directly crash if there are no such return instructions. By implementing these techniques, our framework addresses the challenges of selectively loading and protecting native methods without kernel modifications.

Last but not least, similar to blocking DEX methods, our framework also provides a graceful termination while blocking native functions. Specifically, ART uses function `SharedLibrary::FindSymbol()` to find the address of the native method going to be invoked. After zero-filling the method, the address of the target methods is retained. Consequently, when the `SharedLibrary::FindSymbol()` function is called,

it links the target method to the zero-filled memory space. This step prevents the native code of these blocked methods from being executed or utilized as payloads of code reuse attacks. Our framework modifies this `SharedLibrary::FindSymbol()` function to launch an Activity of the management app via JNI. This activity displays the necessary information to the user, providing transparency and insights into the blocking process.

4 RELATED WORK

Debloating Android applications. With the performance improvement of Android devices, developers commonly add rich functionalities to their Android applications to cater to different user needs. However, some of these features may be irrelevant or unnecessary for certain users. Installing and running such applications can impact system performance and security. Google has recognized this issue and provided solutions from the developers' perspective. For example, Google provides a static analysis tool, i.e., R8, to detect and remove unused code and resources from apps [7]. Google also allows developers to use App Bundle so that only the code and resources needed for a specific device or feature are downloaded [1].

In academics, researchers also developed a series of approaches to debloat Android applications from various perspectives. For example, Jiang et al. remove dead code of Android applications based on static analysis [13]. Pilgun et al. debloated apps by removing the code not executed during the test [17]. Tang et al. debloated apps at the granularity of Activity, Permission, and Modularity [22]. Xie et al. debloated apps to minimize the bandwidth of mobile networks [24]. Unlike previous works, our approach differs in the following aspects: (1) we perform runtime application debloating, rather than debloating at the installation stage targeting tampering the apk file, (2) we are capable of debloating native libraries in Android applications, not limited to just dex files.

Debloating binary programs. In general, static binary analysis is an undecidable problem [23]. Researchers proposed a series of approaches to identify the code to debloat based on static binary analysis. For example, Agadakos et al. removed the unused code by taking advantage of debug symbols to identify function boundaries, construct library function call graphs, and detect address-taken functions that could be targeted by indirect calls [8]. Landsborough et al. employed a genetic algorithm that in toy programs disabled features in binaries [14]. Qian et al. use training and heuristics to identify unnecessary basic blocks and remove them from the binary [18]. Ghaffarinia and Hamlen used a similar approach based on training to limit control flow transfers to unauthorized sections of the code [11].

Recently, researchers focus on removing the unused code in shared libraries across platforms. For example, Mulliner et al. removed unused code in Windows shared libraries (i.e., DLLs) [16]. To build CFGs, they used bounded address tracking to resolve function pointers. Zhang et al. debloated the static library on firmware's shared libraries by erasing the basic blocks not included in the inter-procedural control flow graph (ICFG). They analyzed the global offset table's address loading patterns and decided on all legitimate addresses that could be referenced to build a whole the ICFG [25].

Other works explore the potential of debloating software based on predefined feature sets. For example, TRIMMER finds unnecessary basic blocks using an inter-procedural analysis based on user-defined configurations [20]. CHISEL debloated the program given a highlevel specification from the user [12]. The specification identifies wanted and unwanted program input/output pairs, and requires the source code and the compilation toolchain. To accelerate program reduction, Chisel uses reinforcement learning. It repeats a trial and error approach to make a more precise Markov Decision Process corresponding to the specification.

DamGate is the most related work to our paper [10]. DamGate rewrites binaries with gates to prevent execution of unused features. Different from their work, our work do not modify the binary file itself, instead, we conduct a selectively loading of the native methods during the loading process of the library.

5 CONCLUSION

In this report, we present a novel approach to dynamically block the execution of certain methods within Android applications. We share the list of blocked methods between the Android runtime and the management app by leveraging Android's data-sharing mechanism. We can block both the DEX code compilation and the native code loading. Moreover, we design a graceful termination to improve users' experience while the blocked methods are invoked. Our modification on the AOSP 13 does not violate its default SELinux policy, and thus, it reduces the risk of importing additional security issues. It has a variety of usage scenarios, like blocking unused features by users or blocking malicious methods.

REFERENCES

[1] 2023. About Android App Bundles. https://developer.android.com/guide/app-bundle.
[2] 2023. Android framework classes and services. https://android.googlesource.com/platform/frameworks/base/.
[3] 2023. Android Runtime (ART) and Dalvik. https://source.android.com/docs/core/runtime.
[4] 2023. bionic. https://android.googlesource.com/platform/bionic/.
[5] 2023. Content provider basics. https://developer.android.com/guide/topics/providers/content-provider-basics.

[6] 2023. JNI tips. https://developer.android.com/training/articles/perf-jni.

[7] 2023. Shrink, Obfuscate, and Optimize Your App | Android Studio. https://developer.android.com/build/shrink-code.

[8] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*. ACM, San Juan Puerto Rico USA, 70–83. https://doi.org/10.1145/3359789.3359823

[9] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. 2017. DroidNative: Automating and optimizing detection of Android native code malware variants. *computers & security* 65 (2017), 230–246.

[10] Yurong Chen, Tian Lan, and Guru Venkataramani. 2017. DamGate: Dynamic Adaptive Multi-feature Gating in Program Binaries. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation - FEAST '17*. ACM Press, Dallas, Texas, USA, 23–29. https://doi.org/10.1145/3141235.3141243

[11] Masoud Ghaffarinia and Kevin W. Hamlen. 2019. Binary Control-Flow Trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, London United Kingdom, 1009–1022. https://doi.org/10.1145/3319535.3345665

[12] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Toronto Canada, 380–394. https://doi.org/10.1145/3243734.3243838

[13] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. 2018. RedDroid: Android Application Redundancy Customization Based on Static Analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Memphis, TN, 189–199. https://doi.org/10.1109/ISSRE.2018.00029

[14] Jason Landsborough, Stephen Harding, and Sunny Fugate. 2015. Removing the Kitchen Sink from Software. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, Madrid Spain, 833–838. https://doi.org/10.1145/2739482.2768424

[15] Yan Lin, Joshua Wong, and Debin Gao. 2023. FA3: Fine-Grained Android Application Analysis. In *Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications*. 74–80.

[16] Collin Mulliner and Matthias Neugschwandtner. 2015. Breaking Payloads with Runtime Code Stripping and Image Freezing. https://www.blackhat.com/docs/us-15/materials/us-15-Mulliner-Breaking-Payloads-With-Runtime-Code-Stripping-And-Image-Freezing-wp.pdf

[17] Aleksandr Pilgun. 2020. Don't Trust Me, Test Me: 100% Code Coverage for a 3rd-party Android App. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Singapore, Singapore, 375–384. https://doi.org/10.1109/APSEC51365.2020.00046

[18] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. {RAZOR}: A Framework for Post-deployment Software Debloating. 1733–1750. https://www.usenix.org/conference/usenixsecurity19/presentation/qian

[19] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin TS Chan. 2014. On tracking information flows through jni in android applications. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 180–191.

[20] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, Montpellier France, 329–339. https://doi.org/10.1145/3238147.3238160

[21] Mengtao Sun and Gang Tan. 2014. Nativeguard: Protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. 165–176.

[22] Yutian Tang, Hao Zhou, Xiapu Luo, Ting Chen, Haoyu Wang, Zhou Xu, and Yan Cai. 2021. Xdebloat: Towards automated feature-oriented app debloating. *IEEE Transactions on Software Engineering* 48, 11 (2021), 4501–4520.

[23] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. 2011. Differentiating Code from Data in x86 Binaries. In *Machine Learning and Knowledge Discovery in Databases (Lecture Notes in Computer Science)*, Dimitrios Gunopulos, Thomas Hofmann, Donato Malerba, and Michalis Vazirgiannis (Eds.). Springer, Berlin, Heidelberg, 522–536. https://doi.org/10.1007/978-3-642-23808-6_34

[24] Qinge Xie, Qingyuan Gong, Xinlei He, Yang Chen, Xin Wang, Haitao Zheng, and Ben Y. Zhao. 2023. Trimming Mobile Applications for Bandwidth-Challenged Networks in Developing Regions. *IEEE Transactions on Mobile Computing* 22, 1 (Jan. 2023), 556–573. https://doi.org/10.1109/TMC.2021.3088121

[25] Haotian Zhang, Mengfei Ren, Yu Lei, and Jiang Ming. 2022. One size does not fit all: security hardening of MIPS embedded systems via static binary debloating for shared libraries. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne Switzerland, 255–270. https://doi.org/10.1145/3503222.3507768