Honey, I Shrunk Your App Security: The State of Android App Hardening*

Vincent Haupert[1(✉)], Dominik Maier[2], Nicolas Schneider[1], Julian Kirsch[3], and Tilo Müller[1]

[1] Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany
vincent.haupert@cs.fau.de
[2] TU Berlin, Germany
dmaier@sect.tu-berlin.de
[3] TU Munich, Germany
kirschju@sec.in.tum.de

Abstract. The continued popularity of smartphones has led companies from all business sectors to use them for security-sensitive tasks like two-factor authentication. Android, however, suffers from a fragmented landscape of devices and versions, which leaves many devices unpatched by their manufacturers. This security gap has created a vital market of commercial solutions for *Runtime Application Self-Protection* (RASP) to harden apps and ensure their integrity even on compromised devices. In this paper, we assess the RASP market for Android by providing an overview of the available products and their features. Furthermore, we describe an in-depth case study for a leading RASP product—namely *Promon Shield*—which is being used by approximately 100 companies to protect over 100 million end users worldwide. We demonstrate two attacks against Promon Shield: The first removes the entire protection scheme statically from an app, while the second disables all security measures dynamically at runtime.

1 Introduction

Mobile platforms based on the Google Android and Apple iOS operating systems (OSs) have matured in recent years. They are now omnipresent and form a part of our daily lives. In contrast to desktop platforms, however, some vendors still consider their mobile devices as embedded platforms without rolling security updates. Particularly on Android, device owners may face a problem when the manufacturer leaves a device unpatched, or at least vulnerable, for a long time [33,29,17]. Even in the case where an OS is fully updated, recent history has taught us that new attack vectors can still be uncovered: Rowhammer [27,14] and Spectre [15] are prominent examples of this. Vulnerable end-user devices lead to a challenging situation for companies that promote security apps—such as

* Authors' version of the paper published in the Proceedings of the 15th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2018). DOI: 10.1007/978-3-319-93411-2_4

two-factor authentication apps—which rely on the integrity of the underlying OS. Instead of changing the business logic, for example, by shipping secure hardware tokens or leveraging a phone's *trusted execution environment* (TEE), we noticed a trend toward applying app-hardening solutions that are purely software-based. These solutions are frequently referred to as *Runtime Application Self-Protection* (RASP), and they have already given rise to a vital market.

To the best of our knowledge, our work is the first to challenge the claims of the RASP market by assessing the features commonly offered by RASP products. Furthermore, we provide a detailed security analysis of an internationally leading product called *Promon SHIELD*, which is currently used in more than 30 apps worldwide. In general, we find that RASP solutions cannot protect against their own threat model. In particular, we developed a tool called NOMORP, which is able to automatically disable all security measures employed by Promon Shield.

2 App Hardening

In this section, we discuss the features commonly offered by app-hardening solutions. App hardening, and RASP in general, aim to ensure the security of an app even on a hostile or breached OS. Even though most products are available for Android and iOS, we focus on Android in this work, as many implementations are highly specific to the OS and the ecosystem. We chose Android because of its larger market share and the significant fragmentation of Android versions, both of which lead to a greater demand for app hardening.

App-hardening products are typically provided as *software development kits* (SDKs) with binary libraries, or sometimes as build environment patches that offer the automated integration of security features into an app without the assistance of developers. A central part of an app-hardening solution is obfuscation as it stalls reversing and cracking for a certain period. This is adequate in the gaming industry, for example. In the case of security apps like two-factor authentication apps and financial apps, however, merely increasing the period of protection is not enough. So, app-hardening solutions enrich obfuscation with a variety of defenses against dynamic analysis and best practices.

2.1 The RASP Market

In the course of this research, we analyzed the feature set of 10 commercial app-hardening solutions [9]. The initial goal—to give definite answers about the strength of the provided features—was quickly dismissed, for app-hardening solutions differ vastly per license and app. Some apps protected by the same RASP solution have features enabled that others do not have. Similarly, we noticed apps bearing hardening features in addition to the RASP provider. For all these reasons, the following overview builds on available marketing documents of the RASP products rather than manual analyses of their features.

Table 1 lists market-leading hardening solutions and compares their official feature sets. Additionally, Promon Shield was taken as a case study for in-depth

Product	Anti-Tampering	Anti-Hooking	Anti-Debugging	Anti-Emulator	Code Obfuscation	White-Box Cryptography	Device Binding	Root Detection	Anti-Keylogger	Anti-Screen Reader	Data Encryption	Secure Communication
Arxan for Android	✓	✓	✓	.	✓	✓	.	✓	.	.	✓	.
DNP HyperTech CrackProof	✓	.	✓	✓	.	.	.	✓	.	.	.	.
Entersekt Transakt	✓	.	.	.	.	.	✓	✓	✓	.	.	✓
Gemalto Mobile Protector	.	✓	✓	.	✓	.	✓	✓	✓	.	✓	✓
GuardSquare DexGuard	✓	✓	✓	✓	✓	✓	.	✓	.	.	.	✓
Inside Secure Core for Android	✓	.	✓	.	✓	✓	.	✓	.	.	.	.
Intertrust WhiteCryption	✓	.	✓	.	✓	✓	✓	✓	.	.	.	.
PreEmptive DashO	✓	.	✓	✓	✓	.	✓	✓	.	.	.	.
Promon SHIELD	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SecNeo AppShield	✓	.	✓	.	✓	.	.	.	.	.	✓	.

Table 1: Overview of RASP products and their advertised features.

manual reverse engineering, as discussed in later sections. While investigating Promon Shield, we noticed features that were not mentioned in the official documentation. In other words, a feature not being listed in Table 1 does not necessarily imply that a feature is not present.

Anti-Tampering. If attackers manage to alter the code at runtime, they acquire the same privileges as the underlying app. They can then manipulate data that gets exchanged in the backend, or disable license and security checks. Anti-tampering solutions use a variety of methods to ensure that a third party did not alter an app's code. If tampering the code is possible, other dynamic defenses can be patched out. Therefore, anti-tampering technology is the cornerstone of current app security solutions, and every library in the set offers it.

An obvious anti-tampering approach is to check the signature of the *Android application package* (APK) at startup. If the signature does not match the expected developer certificate, then a third party likely altered the app. More sophisticated measures scatter signature checks throughout the app or use watermarks, thus making the check itself harder to strip [22].

Promon Shield assesses the integrity of the installed `base.apk` by loading the developer's certificate from an encrypted entry in a configuration file at startup. In addition to verifying the APK, it checks the hash sums of certain Android-specific files—such as `AndroidManifest.xml` and `classes.dex`—and its own native library, `libshield.so`.

Anti-Hooking. Even if the code inside the app has not been altered, an attacker can still run code within the app by hooking certain calls and inserting functionality at runtime. The hooking of functions, *Application programming interfaces* (APIs), or system calls allows attackers to modify many elements. For example, they can alter the method parameters and the return values of calls or completely swap out methods. On Android, a variety of open source hooking frameworks exists. Since hooks are usually inserted from the outside, potentially at a higher privilege level, detection is not easy. Anti-hooking solutions for Android typically attempt to find traces of well-known hooking frameworks in the file system or memory. Promon Shield, for example, checks for artifacts of *Xposed* and *Cydia Substrate* but does not scan the app's memory. Therefore, our analysis based on *Frida* remained undetected.

Anti-Debugging. Like in hooking, debuggers can alter the control flow and change the return values of functions that have not been tampered with otherwise. Furthermore, debugging gives attackers insights into the operation of the hardening framework and app. For this very reason, app-hardening solutions try to detect the presence of a debugger and abort the execution if needed. Android has two worlds where a debugger can attach: the native code debugger based on `ptrace` for C and C++, and the Java debugger based on the `JDWP` protocol.

For the Java part, a trivial implementation merely checks if the `isDebuggerConnected` API returns `true`; if it does not, then the execution is aborted. Promon Shield modifies native code data structures to prevent `JDWP` debugging. In the `JdwpState` struct, it replaces the function pointers that are responsible for handling debugger packets with a pointer to a function that always returns the value `false`, leading to the immediate termination of unaltered debugger sessions. To hinder the debugging of native code, libraries usually trace their own code path using `ptrace`. The `ptrace` API allows only a single debugger at a given time. Therefore, if the hardening library already debugs the binary, no other debugger can attach to it. In Promon, the main Promon Shield process SHIELD−1 forks a child process SHIELD−2, which then attaches to all threads of its parent via `ptrace`.

Anti-Emulator. Running an app inside an emulator, a virtual machine, or sandbox allows an attacker to hook or trace program execution. For apps running inside an Android emulator, it is easy to inspect the state of the system, reset it to a saved image, or monitor how the app operates. Several mechanisms for detecting sandboxes are known to be used by malware [18,19,28], and RASP solutions usually implement a subset of the same mechanisms. Promon Shield, for example, immediately crashes the app if it runs inside an Android emulator.

Code Obfuscation. The Android toolchain comes with *ProGuard*, an obfuscation tool that renames all class and method names at compile time. As renaming is a well-known obfuscation method, the techniques used to deobfuscate it have been well researched [3]. Obfuscation, in general, tries to obscure code as much as possible, as do app-hardening solutions. Besides renaming, some other obfuscation

methods are, for example, control-flow flattening, opaque and random predicates, and function merging and splitting [5].

While perfectly obfuscating code running on the same machine as the attacker is impossible according to Barak et al. [1], it can notably increase the effort an attacker has to invest. Krügel et al. state that "[o]bfuscation and de-obfuscation is an arms race [...] usually in favor of the de-obfuscator" [16]. This statement still holds true, as Schrittwieser et al. discussed fairly recently [24].

Another category of obfuscation is DEX packing that aims at mitigating static and dynamic analysis and is particularly popular among malware authors to hide their malicious code from both automatic and manual analysis, e.g., an automated sandbox or a reverse engineer, respectively. The popularity of Android packers is unbroken; hence, research has proposed various approaches for automatic unpacking [31,32,30,6].

The native part of Promon Shield is encrypted and the unpacked code is obfuscated further. The Java part of the `no.promon.shield` package is slightly obfuscated, depending on the target app. If it is, then all packages, classes, methods, and fields are renamed to random eight-character strings. However, Promon Shield does not obfuscate the customer's app using any of the obfuscation techniques mentioned above; at most, *ProGuard* is used independently.

White-Box Cryptography. Like code obfuscation, white-box cryptography aims to obscure secrets [23]. It does not obfuscate the business logic, but tries to prevent cryptographic secrets from leaking [4]. Implementations try to provide a one-way function, making it easy to apply cryptographic operations but hard to reverse the input keys. Like in the case of code obfuscation, attackers can reverse most implementations [10]. While white-box cryptography makes reverse engineering of the key significantly harder, an attacker can still copy the whole implementation blob and apply the cryptographic operations without having to learn the keys used.

Device Binding. Device binding does not prevent copying but stops execution on other devices. For apps like banking, it is desired that they work only on a device that is explicitly paired with the account. So, to comply with the demand of physical second factors, app-hardening solutions try to achieve device binding. They usually fingerprint the device and then store unique identifiers at the first start of the app. If any identifier does not match on app start, the app refuses to run. A common approach is to use the `ANDROID_ID` and `IMEI` as they are unique and very robust [11]. The disadvantage of device fingerprinting compared to approaches that leverage, for example, the hardware-backed key store, is that they rely on information that is accessible to any application running on the same device [2]. In Promon Shield, device binding is rudimentary. It only consists of the `Build.SERIAL` of the device and, if permissions allow it, the IMEI.

Root Detection. On Android, every app runs in its own user context as part of the sandboxing concept. Users who still want to alter certain aspects of their OS often have to *root* their phone, creating a user with elevated privileges. Since rooting breaks the sandboxing concept and allows the user to alter arbitrary

app data, hardening solutions often try to prevent the app from running on rooted phones. Mostly, artifacts of root management applications are checked (e.g., `SuperSU.apk`), or binaries only present on rooted phones (e.g., `/system/bin/su`).

Promon Shield additionally iterates all processes using the `proc` file system and checks for processes like `daemonsu`, belonging to *SuperSU*, a root management app. It also scans `/proc/self/exe`, which resolves to the `app_loader` binary, and scans for *SuperSU* and *Magisk Manager* artifacts. Not only are these checks easy to bypass by renaming the files but they also prevent the execution of protected apps on phones deliberately rooted by their users; however, they cannot prevent privilege escalation exploits on non-rooted phones [7,26].

Anti-Keylogger. The possibility to install third-party keyboard apps on Android opens the system up to malicious keyboards that grab sensitive information entered in the apps. To counteract this, RASP solutions often ship their own keyboards that apps can use in a more trusted fashion. In Promon Shield, if an app uses the provided `SecureEditText` and `SecureKeyboard` classes, Promon's built-in keyboard shows up. As a different line of defense, Promon Shield offers to check the installed keyboard apps against a whitelist. If one of the installed keyboards is not whitelisted in the configuration, the app quits.

Anti-Screen Reader. Malware can use accessibility services, as demonstrated by *Cloak and Dagger* [8], to read the contents on a screen. Anti-screen reader methods try to mitigate these attacks. Promon Shield, for example, iterates through all installed apps that provide accessibility services and checks them against a whitelist provided by the configuration, including their name and signature. Another way to prevent screen content grabbing is by disabling screenshots. When this feature is enabled, Promon Shield sets the `FLAG_SECURE` property on the application's window object at runtime, instructing Android to disallow screenshots and to show a black rectangle in place of media created through the recording API instead of the actual window content.

Data Encryption. App-hardening products sell the idea that data stored encrypted within the app is more secure than data that is encrypted by the system. The idea is to try to prevent attackers with higher privileges from reading stored data. For example, attackers might do this to gain knowledge about possibly sensitive data like transaction histories or encryption keys. The main problem is where the key should be stored so as to remain inaccessible to an attacker. Hence, if not derived at runtime from a user-provided secret, data encryption uses a sort of obfuscation to hide the keys and inner workings as a best-effort solution. Promon Shield provides its customer with the `SecureStorage` class that allows data encryption that also leverages Promon's white-box cryptography. We provide more details on its implementation in Sect. 3.5.

Secure Communication. Secured communication lowers the possibility of man-in-the-middle attacks not only on the network but also against attackers on the phone. Promon Shield has two distinct features for this. First, it offers its own HTTPS networking API for Java. If the app developers switch from using `HttpURLConnection` to `PromonHttpUrlConnection`, all requests get automati-

cally routed through the native lib where certificate pinning is enforced. The native library only connects to servers for which certificates are present in the encrypted configuration file. Additionally, the app developers can choose to add a client certificate to the configuration file in such a way that the server knows whether it is communicating with a genuine app or a third-party client. Second, Promon Shield offers its own protocol based on the `DeviceManagement` class that relies on native methods into `libshield.so`. This Java class allows APIs to register an app at the server and performs signed transactions afterward.

2.2 Threat Model

Based on the diverse mitigations that different hardening solutions claim, we try to synthesize their threat model in this chapter. Hence, we infer our threat model from the one commonly employed by RASP products.

Attacks against Intellectual Property. The first adversary aims at the intellectual property or other secrets, such as the hardcoded credentials, of an app. She has privileged access to her own device and tries to gather insights into the app by means of reverse engineering, including static and dynamic analysis. Her goals are, for example, circumventing licensing checks to pirate an app, cloning the integral functionality of the business logic, or publishing information like hardcoded API keys. She may even be able to offer third-party apps that are fully compatible with the original app by analyzing API calls, leading to potential monetary gain and critical insights into a company's infrastructure.

Attacks against the User Account. The second adversary is a remote attacker who tries to gather user information or execute transactions in the name of the legitimate user. Such attackers usually apply social engineering, drive-by downloads, app piggybacking or any other method to achieve code execution on the victim's device. This attacker is omnipotent as she has access to privilege escalation exploits and can take complete control of the operating system the app is running on. She may trie to run code in the context of the app, communicate to the backend server with the user's credentials, or clone the complete state of the app to her own device for further analysis and use. Additionally, she might use man-in-the-middle attacks to gather sensitive information.

3 Unpacking Promon Shield with Nomorp

Large international finance and public law institutions place their trust in Promon Shield. To demonstrate how this popular RASP product can be thwarted, we propose Nomorp, a tool that automatically disables all protection features from a hardened app. To that end, we developed a static and a dynamic attack to address both threats described in Sect. 2.2. The static version aims at attacking intellectual property, while the dynamic version is after the user's data.

3.1 Promon Shield

Promon, a company from Norway specializing in the security of mobile and desktop applications, is a global player in the RASP market with approximately 100 individual business customers protecting the apps of around 100 million end users [25]. The large number of users, as well as the company's focus on the critical banking sector, makes their product a perfect fit for our in-depth case study analysis. Since Promon does not disclose the names of its customers, we crawled the official Google Play Store for apps using Promon Shield. These are identifiable simply by the inclusion of Promon's characteristic native library, `libshield.so`. After downloading over 150,000 free apps from all Play Store categories, we found 31 apps that include Promon Shield at the time of writing.

Interestingly, even though Promon advertises its solution to other fields—for example, to car manufacturers—all 31 apps in the Google Play Store belong to the finance category. Twenty apps are from Germany, two apps from Norway and Finland, and one app each is from the Netherlands, Sweden, Great Britain, United States, Mexico, Brazil, and Hong Kong. The app's high popularity in Germany is striking; Promon Shield protects most of the banking apps on the German market. As of April 23, 2018, four out of the top ten financial apps in Germany make use of Promon's solution.

The integration process is claimed to be very easy for app developers [20]. After a customer has received Promon's integration tool, developers simply have to specify a configuration file to enable or disable security features. Later, the Promon integration tool takes the app's APK and the configuration file and outputs a hardened APK with the specified protection mechanisms applied. The customer can now publish the resulting APK to the Google Play Store, and no further steps are needed.

The life cycle of an app protected by Promon Shield is illustrated in Fig. 1. The app first loads the native library `libshield.so`. For this, the integration tool adds initializing Java code to the `onCreate` method of the main activity, as specified in `AndroidManifest.xml`. The native library relies on three files which are a product of the integration tool and are added encrypted to the assets of the APK: `mapping.bin`, `config-encrypt.txt`, and `pbi.bin`. After the initialization routine has decrypted and parsed the configuration file, Promon Shield starts a series of threads that realize the enabled features—for example, the anti-debugging or root detection. The configuration defines how Promon Shield should treat anomalies, by allowing the execution of callbacks or directly crashing the app with the possibility to open a web browser with a given URL.

3.2 Static Nomorp

In this section, we propose the use of NOMORP, a fully automated tool intended to strip Promon Shield's app hardening from all apps. We had access to neither the Promon Shield integration tool nor any insider information or internal source code. An adversary could leverage the same tooling and knowledge which makes the tool and analysis particularly relevant. To create NOMORP, we analyzed

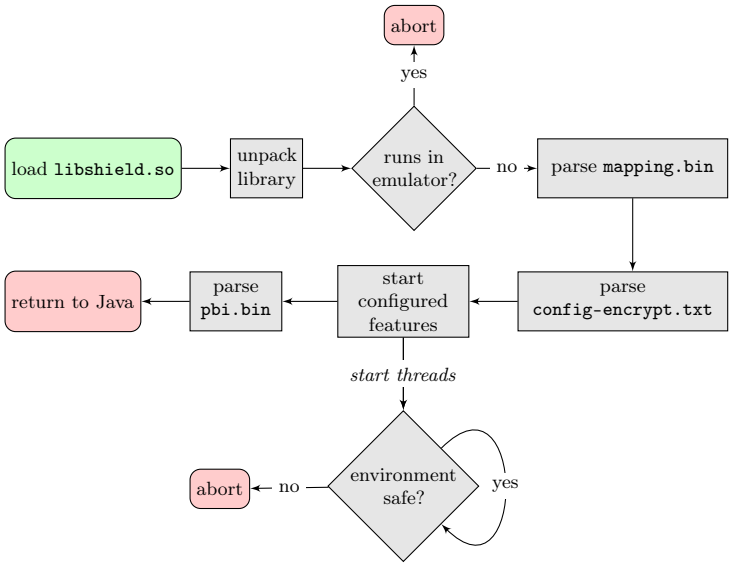

Fig. 1: Life cycle of Promon Shield's native library.

multiple apps that were hardened by Promon Shield. We combined static reverse engineering with dynamic analysis based on a custom Android runtime (ART) and the *Frida* dynamic instrumentation tool. Our analysis quickly revealed that Promon Shield adds a native library to the app that gets loaded first. As stated, this library is easily detected, for the naming of the file always follows the pattern `libshield_{ID}.so`. We leveraged this knowledge to detect additional apps in the Google Play Store that use Promon Shield. As described in Sect. 3.5, we used this library and all calls from Java to it as a starting point in the analysis. After discovering that Promon left the Java part largely untouched, we focussed on either removing `libshield.so` statically or disabling it dynamically at runtime. We ended up implementing both methods in NOMORP. This section presents sNOMORP, a tool that is capable of producing a version that is easier to analyze. In Sect. 3.3, we describe dNOMORP, a tool to disable Promon Shield at runtime.

The native library `libshield.so` implements most app-hardening features internally. Stripping it automatically disables almost all security measures. Promon is aware of the attack scenario and seeks to prevent it by introducing a binding between the customer's Java code and their own native code implemented in `libshield.so`. This mainly consists of two mechanisms: externalization of strings, and externalization of constants. Hereinafter, we explain how Promon Shield implements each feature and how we circumvented them.

String Externalization. Promon Shield's string externalization is visualized in Fig. 2 and works as follows: When the integration tool applies Promon Shield's protection, it looks for strings inside the client's Java code. For each string, it creates an entry in an index-string dictionary with a linear increasing index.

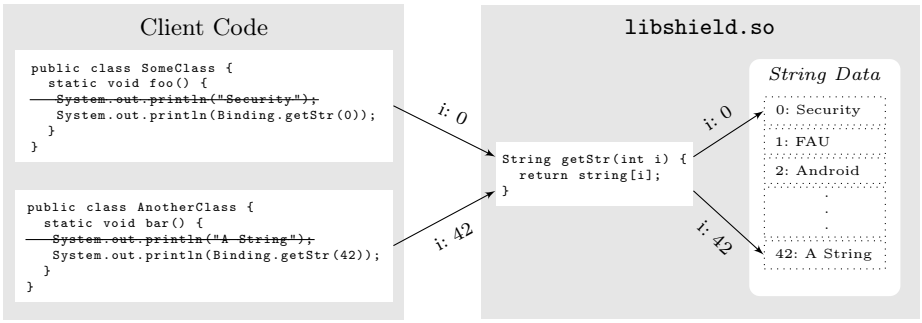

Fig. 2: Visualization of Promon Shield's string externalization.

This also applies to strings of the same value. The string declaration is then removed from the byte code and replaced by a method invocation that points to `libshield.so`. This method takes an integer as an argument and returns the corresponding string value whenever it gets called with a key of the previously created index-string dictionary. Due to its semantics, we dubbed this function `getStr`. For example, a previous declaration of the Java string `"Hello World"` gets replaced by a call to `getStr(123)`, which returns `"Hello World"` as a result.

Constant Externalization. Apart from the substitution of strings with method calls to Promon's `getStr` method, it also externalizes Java constants in a clever way. This works as follows and the process is further illustrated in Fig. 3: 1) The Promon integration tool replaces any class member field declared as `static` and `final` with a random value of the correct type. Similar to the string externalization, the replaced values of a class are stored in a nested dictionary: While the first level takes the fully qualified class name as a key, its value is a dictionary that maps the constant names of the class to their original value, including the type. To restore these values prior to the first use of the class, Promon Shield adds a call with the `Class` object as an argument to its Java glue code in the static constructor of the class. The ART executes the static constructor as soon as the class gets loaded, enabling execution of the code even before a static field is accessed. 2) The Java wrapper code of Promon Shield replaces the dots in the fully qualified class name as returned by `Class.getName()` with slashes and invokes the corresponding native method. The string alternation likely remains compatible with the previously created dictionary. 3) Inside `libshield.so`, the method looks up the class name and reads out all the key-value pairs for this class. 4) In the last step, the native code replaces the random constant value with the original values by means of reflection. Given the way the method works, it is dubbed `pushToClass`. Notably, it is not possible to dynamically alter the value of fields declared as `final` using Java. The Java Native Interface (JNI), however, does not have this restriction.

Rewriting the App. To rewrite the app using *dexlib2*, we have to apply the mappings of the Promon integration tool in reverse. The trivial and straightfor-

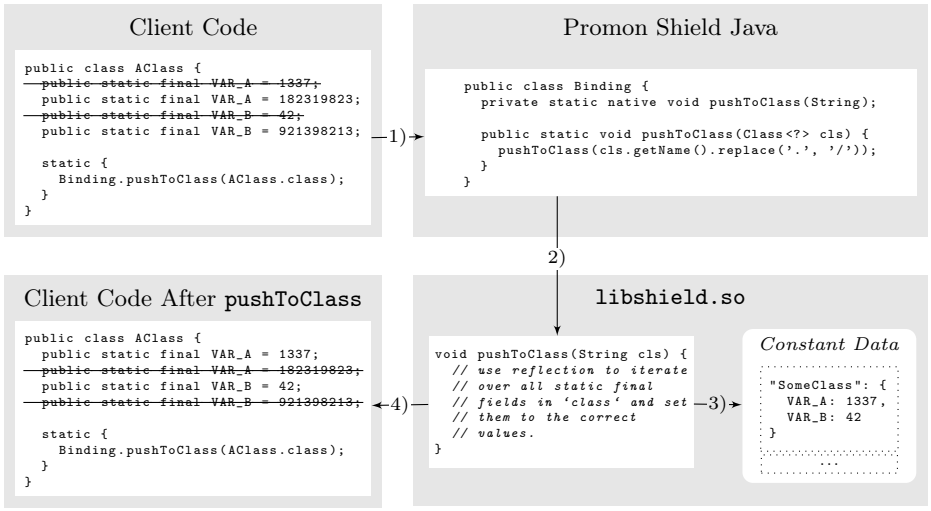

Fig. 3: Visualization of Promon Shield's constant externalization.

ward method for string externalization is to statically determine the highest index N for the argument of the `getStr` method. Next, we use *Frida* to dynamically invoke `getStr` with each value in the range $[0; N]$, thus creating an index-string mapping. The required mapping for the constant obfuscation can be created similarly: After determining all the classes with a static constructor that calls `pushToClass`, we dynamically access these classes. Alternatively, we could just iterate all the classes. The resultant mapping is sufficient to assign the original values to the constants.

A subtle detail that hindered the straightforward usage of *dexlib2* to rewrite the DEX bytecode of the app was the renaming obfuscation of Promon Shield's own Java code. As the renaming always differed between apps and even between versions, we would have needed heuristics to identify the class and method names we wished to rewrite. In the course of our analysis, however, we realized that the hash sum of the bundled `libshield.so` did not always differ between two different apps, whereas the class and member names of Promon's code did. This indicates not only that Promon ships their native library `libshield.so` precompiled to their customers but also that Promon Shield requires a renaming mapping to make its native methods accessible from Java code by calling `RegisterNatives`. Furthermore, it implies that `libshield.so` has mappings for the string and constant externalization.

Through dynamic analysis based on *Frida*, we acquired all the mappings in plain text, including the configuration file. We have been successful with a combination of hooks to `malloc`, `free`, and `memset`. On each execution of `malloc`, we add the returned pointer to an internal list that our algorithm traverses on each execution of `free`. We were surprised that this approach succeeded, as we expected the native library to contain its own dynamic memory management

implementation. This becomes even more significant because Promon seems to be aware of the attack surface, as it uses `memset` to clear (at least some) buffers before freeing them. To ensure that we do not miss any buffer, our code also walks the list of buffers before each execution of `memset`. For performance reasons, we implemented our *Frida* hooks in C instead of JavaScript. This happens through our own native library `libnomorp.so`, which gets loaded first and particularly before `libshield.so`. With this approach, we were able to quickly retrieve the plain text configuration file and the two mappings. The file `config-encrypt.txt` contains the customer-defined configuration file in the CSV format, while `mapping.bin` and `pbi.bin` are both JSON dictionaries: The first stores information about the client code's string and constant obfuscation, as explained earlier, and the second contains a renaming mapping of Promon Shield's Java code obfuscation—for example, its original class and method names.

Evaluation. Using sNomorp, an adversary can produce an app version that is easier to analyze statically. The entire process is fully automated and takes no longer than five minutes from the start of the download of the APK until the output of the rewritten app. Apart from that—even though not a declared goal of the applied threat model—the majority of apps processed using sNomorp is even fully runnable after our tool stripped Promon Shield from the app.

During the large scale analysis, however, 9 Apps used a different version of Promon Shield that registers the device at the backend from inside the library. Stripping the library completely breaks the HTTP communication channel. This means that sNomorp is less useful in this case, because the HTTP requests are performed from within `libshield.so`. Dynamic analysis of the communication is, however, possible using dNomorp that we present in Sect. 3.3.

3.3 Dynamic Nomorp

While the previous approach, sNomorp, explained how to remove Promon Shield from an app, this section is dedicated to a dynamic procedure; hence, we call it dNomorp. In contrast to sNomorp, we keep using `libshield.so` but disable all its features at runtime.

To alter the execution dynamically, we still need to insert our custom native library in order to hook functionality. Similar to sNomorp, this library extracts the configuration file and the mapping. Instead of dumping them for further analysis, however, dNomorp then rewrites the configuration file on the fly as soon as it gets decrypted. This clear text configuration file consists of key-value pairs separated by a semicolon.

In the configuration of Promon Shield, most security features have three entries: a binary value indicating if the features are enabled, another binary value indicating if the app should crash, and a third value that may contain a percent-encoded URL that would be opened if the app crashes. For app repackaging, these entries may look like `checkRepackaging=1`, `exitOnRepackaging=0`, and `exitOnRepackagingURL=https%3A%2F%2Ffau.de`.

The objective of our attack is to disable all features of Promon Shield by replacing values of `1` with `0` at loading time using a hook that rewrites the plain

Fig. 4: Visualization of rewriting Promon Shield's configuration at runtime.

text configuration file after decryption but before evaluation. The parsing of the configuration file consists of four steps, which are shown in Fig. 4:

1) Read the encrypted content of the file `config-encrypt.txt` from the APK and write it to a `malloc`-allocated buffer.
2) Decrypt the configuration file using Promon Shield's white-box cryptography. The result is once again written to a dynamically allocated buffer at the heap.
3) The native library reads the cleartext configuration file.
4) The data gets parsed into the library's internal data structures.

Right after the configuration file has been decrypted in 2), it is copied to another buffer using `memcpy` before `libshield.so` starts to read the decrypted content in 3). To that end, we installed a hook to `memcpy` just before the content of the old buffer gets copied into the new one. By modifying the source buffer in our `memcpy` hook before delegating to the real `memcpy` function, we can effectively modify the content of the target buffer and, therefore, the configuration file (Step 2.1) in Fig. 4). Obviously, `memcpy` is a frequently used function, and its use is not just limited to the configuration parsing. For that reason, we search the buffers for well-known configuration entries like `checkDebugger` and `checkRooting`.

Evaluation. In contrast to the static attack described earlier, the dynamic configuration rewrite attack is straight-forward. We used the fact that Promon Shield is commercial software with a licensing model and the intent to be configurable without having to recompile the binary blob every time. Instead of manually disabling each function individually—like we had to do for sNomorp—we ask the library to disable the features using their intended configuration.

dNomorp works very well and reliably on all the 31 apps we identified to use Promon Shield. In contrast to the static approach, dNomorp does not help static analysis, for it does not internalize the constants and strings as they are defined in `pbi.bin`. In return, however, the dynamic approach produces an APK that is fully compatible with the original version where Promon Shield is still part of the app, including all of its own functionality. Values secured by the white-box cryptography inside `libshield.so`, for example, can still be used. An attacker

can now modify the app at will (e.g., add malicious code) or dynamically analyze it (e.g., run a man-in-the-middle attack against the network communication).

3.4 Coordinated Disclosure

We informed Promon about our results in close cooperation with Hakan Tanriverdi, a journalist at *Süddeutsche Zeitung* who frequently writes articles in the area of information security. He first made contact with Promon and later asked the affected German financial institutions for their opinion on the case. Finally, Mr. Tanriverdi published an article [25] about our findings in the business section of Süddeutsche Zeitung on November 24, 2017, describing our attack in an abstract way without disclosing technical details. We decided to not disclose any of our source codes to third parties, in order to avoid supporting cybercriminals.

When we let Promon know that our proposal to present the attacks at the *34th Chaos Communication Congress (34c3)* had been accepted, we agreed to not disclose any further information before the 34c3 talk. Additionally, we provided Promon with a detailed description of the weaknesses of their RASP product. On December 27, 2017, we presented our attack to the audience at 34c3.

3.5 Inside Past and Present Libshield.so

This section is dedicated to the internals of `libshield.so`, the core part of Promon's hardening solution. In reaction to our findings, Promon introduced some modifications to its native library and we describe these changes at the end of this section. At the time of writing, however, large parts of our past analysis still apply to recent versions of Promon Shield.

During normal mode of operation, Promon Shield uses a variety of cryptographic primitives to secure the contents of `libshield.so`: several files in the `assets` directory, and the contents of the secure storage feature Promon offers. Figure 5 shows an overview of `libshield.so` and depicts the decryption of the mapping file `pbi.bin`. Promon, however, processes the other files similarly.

Executable. Encryption protects various sections of the shared library `libshield.so`. After loading the image into memory and resolving all its dependencies, the dynamic loader dispatches the constructors specified in `.init_array`. One constructor invokes an obfuscated version of the RC4 cipher to decrypt the `.rodata`, `.text`, and `.ncd` sections of the binary employing three distinct keys that are specific to the version of Promon Shield used.

Assets. As part of the integration process of Promon Shield into the target app, Promon creates three files and stores them encrypted in the `assets` folder of the app: `config-encrypt.txt`, `mappings.bin`, and `pbi.bin`. Section 3.2 explains the purpose of each file. The initialization procedure of `libshield.so` applies the following steps to these files to decrypt them: First, a SHA512 hash h' over all but the last 64 bytes is calculated. Second, a fixed DER-encoded public RSA key pk is generated using the same combination of arithmetic operations and loops that conceal all other constant strings needed. This public RSA key is used

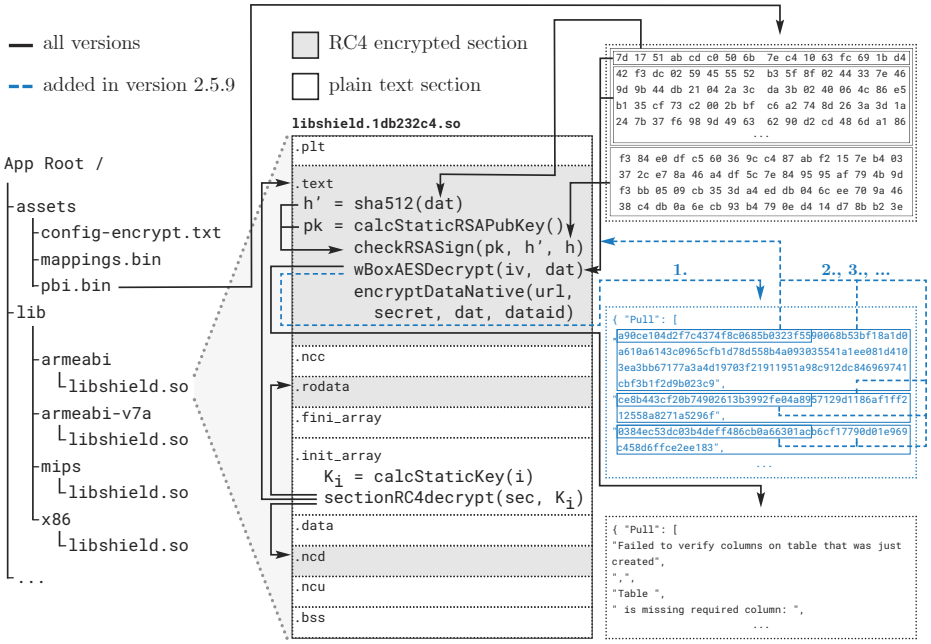

Fig. 5: Cryptographic functions protecting `libshield.so` and selected assets.

afterwards to decrypt the last 64 bytes h of the file, to allow the comparison of the result and h'. This third step effectively constitutes a signature validation, and is only decrypted using a white-box implementation of AES128 if h and $RSA_D(h', pk)$ match the contents. The white-box relies on OpenSSL and a custom implementation of the block cipher that uses the secret symmetric key in an expanded version only. AES128 is applied in CBC mode with the first 16 bytes of the data serving as initialization vector (IV). The result is expanded using the `inflate` algorithm of `zlib`.

Secure Storage. Applications can request Promon Shield to encrypt data through the Java class `SecureStorage` which internally uses `libshield.so`'s native function `encryptDataNative`. Apart from the data requested for encryption (`dat`), the function requires a URL parameter of a Promon Shield server endpoint (`url`), and two byte strings: one chosen randomly at runtime (`secret`), and another one to identify the data to store (`dataid`). Promon Shield first applies an HMAC to `dataid` using `secret` as the HMAC key and stores the result in `dataid`. This step is repeated 4096 times in total before calculating a 16 byte hard-coded device identifier (again using the obfuscation technique described earlier). Together with a `protocol` and a `msg` variable, as well as the version of Promon Shield, the device and data id serve as HTTP form data (`application/x-www-form-urlencoded`) that `libshield.so` submits via HTTPS (with an optional TLS client certificate). After successfully transferring

this information to the server specified by the `url` parameter, the backend responds with a key specific for the combination of all parameters used in the corresponding request. This key is then used for AES256 in CBC mode to encrypt the data specified by `dat`.

Changes Introduced in Later Versions of Promon Shield. In response to our attacks, Promon introduced a few countermeasures. To the best of our knowledge, the changes were introduced in version 2.5.9 and all apply to Promon Shield's handling of the encrypted configuration `config-encrypt.txt` and the push and pull bindings contained in `pbi.bin`. Promon's renaming mapping for its Java obfuscation, `mapping.bin`, remains untouched.

First, Promon Shield now uses two layers of its AES white-box for both files to prevent revealing the configuration's location via easy to recognize strings in memory. For this purpose, all configuration keys have additionally been replaced by a combination of 16 + 4 byte identifiers. Second, the configuration parameters indicating whether Promon Shield should perform certain runtime checks—for example `checkRooting`—were removed and are now directly compiled into the code of `libshield.so`.

Altogether, these modifications to Promon Shield aim at making NOMORP stop working. Even though neither sNOMORP nor dNOMORP function for recent versions of Promon Shield, we are confident that the required adjustments are minor. To substantially increase the effort an adversary has to take, Promon should implement further improvements as suggested in the following section.

4 Discussion and Improvements

In this section, we discuss the findings and show what RASP providers can do to improve their offerings. Attackers will always have an advantage over the defending RASP solution as they only need to find single failures and can look at the implementation statically. App developers therefore should consider two-factor authentication and server-based solutions where possible.

In the following, we use the term *RASP provider* for developers of RASP solutions, *customer* for app developers using RASP to secure their apps and *end user* for the user of an app developed by customers. Our treat model outlined in Sect. 2.2 implies the following goals for a RASP provider: First, make analysis harder and more expensive so that an adversary cannot easily steal the customer's nor the RASP provider's intellectual property (IP). Second, mitigate automated large scale attacks against the customers' products.

With increased value of the protected contents, the motivation to break the protection increases; hence, defense mechanisms which are sufficient for one app might not be enough for another. The possibility to steal money, for example, increases the literal payout of successful attacks. The more valuable a successful attack is, the stronger RASP protection needs to be.

Of course, with more customers using a certain RASP implementation in their apps, the value of a generalized attack against a RASP solution increases. This does not only demand that RASP providers continuously improve their

libraries, but also calls for diversification of their product on a per customer per app basis. On top, randomness should be introduced in the build steps to make sure all updates of an app look different. Another main way to hinder automated unpacking is to interweave the code of the RASP provider and the customer tightly. Both measures, individualization and interweaving the RASP providers and the customer's code, would frustrate reverse engineering and mitigate automatic attacks. In the following, we give more detailed suggestions on how to strengthen the security of RASP solutions.

Avoid Easy to Track Resource Files. The NOMORP unpacker was very effective once we found out at which point the configuration was processed. At that point, our tool could disable features by simply altering the configuration. RASP solutions should never offer options to deactivate features at runtime as an attacker can always leverage them for their benefit. Instead of loading a configuration as asset that can be traced, RASP providers should not include features in the binary that the customer disabled. On the other hand, activated features should not have single points where an adversary can disable them.

In addition to the configuration file, we could recover the mappings of obfuscated strings and constants by tracing accesses to the asset files. Compiling the mappings into the main binary blob at random positions would already have increased the attack complexity.

Since an attacker can still iterate over all elements and recreate the mappings dynamically, a RASP solution could call the next JNI functions directly instead of returning the value [21]. For this purpose, a function for each constant could be created in native code, taking additional parameters forwarded to the next function call and then filling in the needed parameter with the constant value.

Anti-Tampering. Measures to assert the integrity of the customer and RASP code are already widely applied. A good defense checks the integrity of certain code blocks often and in different ways to make it harder to disable or alter it.

Anti-Hooking & Anti-Debugging. Anti-debugging and anti-hooking are close relatives. An attacker can replace debugging tasks like reading from memory using hooks. Likewise, if an app allows debugging, adding hooks is trivial. As for anti-hooking, RASPs already implement detections for hooks through `LD_PRELOAD`, *Xposed* or *Cydia Substrate* and *Frida*, however are not yet resilient against simple changes in those frameworks. Generally speaking, RASP providers should employ more general detection mechanisms. Adding more runtime integrity checks, raising alarms from obfuscated code, checking they have actually fired after the anticipated time and more can add enough complexity to make hooking and debugging a burden. Code obfuscation can help obscuring the location of anti-hooking and anti-debugging checks. Most importantly, the RASP provider should interweave their code and the included checks with the client's code so that no obvious interfaces are exposed that an adversary can disable, e.g., on startup. Similarly, once the integrity is breached, the app should fail fast. Since callbacks could be hooked to keep the app running, crashing the app without an explicit exit call or any status report is the safest option.

Anti-Emulator. If the app runs inside an emulator, debugging and hooking remains trivial. A perfect emulator can hardly exist, so emulator fingerprinting can add manual work to the analysis. For this, many environment checks can be added at random positions in the code [18]. Research in automatic unpacking of malware samples based on whole-system emulation suggests that anti-emulation techniques are effective [6].

Device Binding. To mitigate running a cloned app on a different device that was used for registration, an app can make use of device binding. For this, aggressive fingerprinting can be employed, aggregating multiple environment values. On modern hardware apps should create asymmetric keys in the trusted environment, if available. From there, keys can hardly be extracted. Adding a signature to the network configuration using this key lets the server know it is still talking to the same device.

Code Obfuscation. RASP should not only obfuscate and strip its own but also the customer's code. The larger and more interwoven the obfuscated code is, the harder it is to reverse. This means that the integration tool of the RASP provider should even include third party libraries to create one entangled unit where possible. Interweaving can be done within native libraries, DEX bytecode and between the native library and Java using JNI. Modern Android specific obfuscation methods like DEX packing and VM-based obfuscation should be considered [32,31]. Another aspect is the randomization of compilation passes. If every version of an app looks different, it will be hard to automate unpackers and to adapt to the latest changes of apps. Some of the steps discussed may require RASP providers to ship source code or intermediate language together with a toolchain to their customers. In turn, it requires that RASP providers value the IP of the customer higher than their own.

Root Detection. Root detection can help to slow down reverse engineering. Many analysis frameworks require root. It does, however, not defend the user against attacks that use privilege escalation exploits. App developers have to decide if they really need to block rooted users or if it is enough to warn the user of possible consequences. As for root detection, relying on the presence of certain files, like `su`, is too easy to circumvent. Instead, two alternatives appear useful:

1) Google SafetyNet is an API that allows apps to check the integrity of the device they are running on. To prevent an attacker from hooking the decision, the implementation must check the attestation result on the server rather than on the client side. This means RASP Providers and app manufacturers need to support this in their backend.
2) Privilege escalation exploits cannot be prevented or detected. The only way to reduce the likelihood is to require a sound minimum version of Android and its security patch level. Instead of comparing a string, the app should rely on a new API to talk to its backend. That way, the app crashes if its communication does not employ the new API.

Anti-Keylogger & Anti-Screen Reader. An internal keyboard or keyboard whitelists are reasonable. An attacker, however, may still overlay it with their own app [8].

Secure Communication. RASP solutions can automatically upgrade TLS communications with certificate pinning. If client and server certificate are obfuscated and hidden at random positions, the effort the attacker needs to invest to build a man-in-the-middle server for analysis and therefore reversing the API is increased. Requiring newer client certificates on the server side often additionally increases the complexity.

Encryption. Storing data encrypted makes it considerably harder for attackers. Newer phones can create and store encryption material securely inside the hardware-backed key store. If attackers is on the system before keys are created, however, they can hook the API calls for key creation and provide their own. A RASP may, for this reason or to support older hardware, choose to employ white-box cryptography. White-box cryptography obfuscates a static key making it more difficult to recover it. An attacker, however, may use the whole crypto mechanism as a black box and decrypt secrets with it. The white-box code should therefore be tightly interwoven with the rest of the code and entry points to the crypto functionality as well as their use should not be obvious. Just like for constants, the RASP solution can call Java functions directly using JNI instead of returning the decrypted secrets from a function. The white-box and the mechanisms to harden it should be changed often and a previous version should only be accepted by the server for a limited period of time.

5 Conclusion

Taking into account the threat model from Sect. 2, we showed, based on a case study of Promon Shield, that RASP solutions do not uphold their security promises. Relying on obfuscation and other software-based hardening techniques cannot replace established security practices like two-factor authentication if the stakes are high, e.g., in financial apps.

During our evaluation, we systematically broke all security guarantees of our case study. Our tool NOMORP is capable of dynamically disabling all security measures of Promon Shield by altering protected apps at runtime. Other frameworks discussed in Sect. 2 were not vetted as thoroughly. While they might make use of stronger obfuscation and hardening we still believe their defenses can be broken one way or another. Similar, fully automated tools can therefore be built for all app-hardening solutions. Application layer security mechanisms will always lose against elaborated attackers [12], since they operate at the same privilege level as the attackers or at an even lower one. Worse, research has shown that RASP providers can even introduce severe security vulnerabilities [13].

As a short-term line of defense, RASP providers can use stronger obfuscation, improve their detection measures for hooking, debugging and rooting and add additional hardening steps to the client app as we discussed in section 4. These

measures make attacks harder but RASP solutions should still not claim security-sensitive apps can be used on compromised devices under all circumstances. Instead, they need to communicate to their clients clearly that they will only raise the bar for attackers for a certain time. Of course, this can be a valid defense, according to the company's threat model. RASP providers need to develop methods that are less scalable through automated attacks by providing individualized solutions for their customers.

For the future of two-factor authentication, the only way forward is to shift the security vectors toward secure tokens in hardware, like TEEs, as well as backend-based fraud detection, instead of relying on solutions that appear good enough but are conceptually flawed.

For the time being, users should make sure to run the latest updates and security patches and to upgrade their mobile devices if they are no longer patched. RASP providers need to develop methods that are less scalable through automated attacks by providing individualized solutions for their customers. While we showed all RASP systems can theoretically be broken, in practice not all is lost: RASP providers will have reached their goals to secure the mobile app market once adversaries do not consider reversing hardened apps worth it as it is simply too complex with very little gain.

Acknowledgments

We wish to thank our shepherd Yanick Fratantonio and the anonymous reviewers for their helpful comments. Furthermore, we appreciate Felix Freiling's support during the disclosure process.

The work presented in this paper was conducted within the research project "Software-based Hardening for Mobile Applications" and was partially funded by the German Federal Ministry of Education and Research (BMBF).

References

1. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S.P., Yang, K.: On the (im)possibility of obfuscating programs. In: Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings. pp. 1–18 (2001)
2. Bianchi, A., Gustafson, E., Fratantonio, Y., Kruegel, C., Vigna, G.: Exploitation and mitigation of authentication schemes based on device-public information. In: Proceedings of the 33rd Annual Computer Security Applications Conference. pp. 16–27. ACSAC 2017, ACM, New York, NY, USA (2017)
3. Bichsel, B., Raychev, V., Tsankov, P., Vechev, M.T.: Statistical deobfuscation of android applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 343–355 (2016)
4. Chow, S., Eisen, P.A., Johnson, H., van Oorschot, P.C.: White-box cryptography and an AES implementation. In: Nyberg, K., Heys, H.M. (eds.) Selected Areas in

Cryptography, 9th Annual International Workshop, SAC 2002, St. John's, Newfoundland, Canada, August 15-16, 2002. Revised Papers. Lecture Notes in Computer Science, vol. 2595, pp. 250–270. Springer (2002)
5. Collberg, C., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley Professional, 1st edn. (2009)
6. Duan, Y., Zhang, M., Bhaskar, A.V., Yin, H., Pan, X., Li, T., Wang, X., Wang, X.: Things you may not know about android (un)packers: A systematic study based on whole-system emulation. In: 25th Annual Network and Distributed System Security Symposium, NDSS, 2018, San Diego, California, USA, February 18 - 21, 2018 (2018)
7. Felt, A.P., Finifter, M., Chin, E., Hanna, S., Wagner, D.A.: A survey of mobile malware in the wild. In: Jiang, X., Bhattacharya, A., Dasgupta, P., Enck, W. (eds.) SPSM'11, Proceedings of the 1st ACM Workshop Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2011, October 17, 2011, Chicago, IL, USA. pp. 3–14. ACM (2011)
8. Fratantonio, Y., Qian, C., Chung, S.P., Lee, W.: Cloak and dagger: From two permissions to complete control of the UI feedback loop. In: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017. pp. 1041–1057 (2017)
9. Gartner, Inc.: Market guide for application shielding (06 2017), `https://www.gartner.com/doc/3747622/market-guide-application-shielding`
10. Goubin, L., Masereel, J., Quisquater, M.: Cryptanalysis of white box DES implementations. In: Adams, C.M., Miri, A., Wiener, M.J. (eds.) Selected Areas in Cryptography, 14th International Workshop, SAC 2007, Ottawa, Canada, August 16-17, 2007, Revised Selected Papers. Lecture Notes in Computer Science, vol. 4876, pp. 278–295. Springer (2007)
11. Haupert, V., Müller, T.: On app-based matrix code authentication in online banking. In: Furnell, S., Mori, P., Camp, O. (eds.) Proceedings of the 4th International Conference on Information Systems Security and Privacy, ICISSP 2018, Funchal, Madeira, Portugal, February 22-24, 2018. pp. 149–160 (2018)
12. Jung, J., Kim, J.Y., Lee, H., Yi, J.H.: Repackaging attack on android banking applications and its countermeasures. Wireless Personal Communications **73**(4), 1421–1437 (2013)
13. Kim, T., Ha, H., Choi, S., Jung, J., Chun, B.: Breaking ad-hoc runtime integrity protection mechanisms in android financial apps. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017. pp. 179–192 (2017)
14. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014. pp. 361–372 (2014)
15. Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. CoRR **abs/1801.01203** (2018), `http://arxiv.org/abs/1801.01203`
16. Krügel, C., Robertson, W.K., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA. pp. 255–270 (2004)
17. Luu, D.: How out of date are android devices? (2017), `https://danluu.com/android-updates`

18. Maier, D., Müller, T., Protsenko, M.: Divide-and-conquer: Why android malware cannot be stopped. In: Ninth International Conference on Availability, Reliability and Security, ARES 2014, Fribourg, Switzerland, September 8-12, 2014. pp. 30–39. IEEE Computer Society (2014)
19. Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.: Rage against the virtual machine: hindering dynamic analysis of android malware. In: Balzarotti, D., Caballero, J. (eds.) Proceedings of the Seventh European Workshop on System Security, EuroSec 2014, April 13, 2014, Amsterdam, The Netherlands. pp. 5:1–5:6. ACM (2014)
20. Promon AS: Shield: Application protection and security for mobile apps, `https://promon.co/products/mobile-app-security`
21. Protsenko, M., Kreuter, S., Müller, T.: Dynamic self-protection and tamperproofing for android apps using native code. In: 10th International Conference on Availability, Reliability and Security, ARES 2015, Toulouse, France, August 24-27, 2015. pp. 129–138 (2015)
22. Ren, C., Chen, K., Liu, P.: Droidmarking: resilient software watermarking for impeding android application repackaging. In: Crnkovic, I., Chechik, M., Grünbacher, P. (eds.) ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014. pp. 635–646. ACM (2014)
23. Saxena, A., Wyseur, B.: On white-box cryptography and obfuscation. CoRR **abs/0805.4648** (2008), `http://arxiv.org/abs/0805.4648`
24. Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., Weippl, E.R.: Protecting software through obfuscation: Can it keep pace with progress in code analysis? ACM Comput. Surv. **49**(1), 4:1–4:37 (2016)
25. Tanriverdi, H.: Überweisung vom Hacker. Süddeutsche Zeitung **73**(270) (11 2017)
26. Thomas, D.R., Beresford, A.R., Rice, A.C.: Security metrics for the android ecosystem. In: Lie, D., Wurster, G. (eds.) Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2015, Denver, Colorado, USA, October 12, 2015. pp. 87–98. ACM (2015)
27. van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., Giuffrida, C.: Drammer: Deterministic rowhammer attacks on mobile platforms. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 1675–1689 (2016)
28. Vidas, T., Christin, N.: Evading android runtime analysis via sandbox detection. In: Moriai, S., Jaeger, T., Sakurai, K. (eds.) 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014. pp. 447–458. ACM (2014)
29. Wu, L., Grace, M.C., Zhou, Y., Wu, C., Jiang, X.: The impact of vendor customizations on android security. In: 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013. pp. 623–634 (2013)
30. Xue, L., Luo, X., Yu, L., Wang, S., Wu, D.: Adaptive unpacking of android apps. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). pp. 358–369 (2017)
31. Yang, W., Zhang, Y., Li, J., Shu, J., Li, B., Hu, W., Gu, D.: Appspear: Bytecode decrypting and DEX reassembling for packed android malware. In: Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings. pp. 359–381 (2015)

32. Zhang, Y., Luo, X., Yin, H.: Dexhunter: Toward extracting hidden code from packed android applications. In: Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II. pp. 293–311 (2015)
33. Zhou, X., Lee, Y., Zhang, N., Naveed, M., Wang, X.: The peril of fragmentation: Security hazards in android device driver customizations. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014. pp. 409–423 (2014)