Dissecting Residual APIs in Custom Android ROMs

Zeinab El-Rewini
University of Waterloo
zelrewin@uwaterloo.ca

Yousra Aafer
University of Waterloo
yousra.aafer@uwaterloo.ca

ABSTRACT

Many classic software vulnerabilities (e.g., Heartbleed) are rooted in unused code. In this work, we aim to understand whether unused Android functionality may similarly open unnecessary attack opportunities. Our study focuses on OEM-introduced APIs, which are added and removed erratically through different device models and releases. This instability contributes to the production of bloated custom APIs, some of which may not even be used on a particular device. We call such unused APIs *Residuals*.

In this work, we conduct the first large-scale investigation of custom Android Residuals to understand whether they may lead to access control vulnerabilities. Our investigation is driven by the intuition that it is challenging for vendor developers to ensure proper protection of Residuals. Since they are deemed unnecessary, Residuals are naturally overlooked during integration and maintenance. This is particularly exacerbated by the complexities of Android's ever-evolving access control mechanism.

To facilitate the study at large, we propose a set of analysis techniques that detect and evaluate Residuals' access control enforcement. Our techniques feature a synergy between application and framework program analysis to recognize potential Residuals in specially curated ROM samples. The Residual implementations are then statically analyzed to detect potential *evolution-induced* access control vulnerabilities. Our study reveals that Residuals are prevalent among OEMs. More importantly, we find that their presence may even lead to security-critical vulnerabilities.

CCS CONCEPTS

• **Systems security → Mobile platform security**.

KEYWORDS

Mobile platform security; access control; software debloating

ACM Reference Format:
Zeinab El-Rewini and Yousra Aafer. 2021. Dissecting Residual APIs in Custom Android ROMs. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460120.3485374

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3485374

1 INTRODUCTION

Never before has any operating system (OS) been so popular and diverse as Android. Over the last decade, phone manufacturers and carriers around the globe have built their devices upon the Linux-based open source platform. Such success, however, does not come without risks. The open source nature of the OS has long enabled the Original Equipment Manufacturers (OEMs) to frequently tailor the existing Android Open Source Project (AOSP) codebases to different custom hardware and services. Despite Google's efforts (i.e., through Treble Project) to regulate Android ecosystem diversity, OEM customization remains highly under-regulated, leading to the inevitable introduction of vulnerabilities and security flaws. Indeed, prominent studies show that OEM introduced functionalities fail to properly protect underlying sensitive resources [11, 32, 37, 38] and that significant numbers of pre-installed apps on custom ROMs are riddled with classic Android vulnerabilities [21, 34].

Of particular interest are the evolution-related vulnerabilities introduced when OEMs cannot respect well-established security requirements while keeping up with the fast pace of Android version updates and the sophistication of new functional requirements. For each new Android version and device model, OEM developers adapt the existing custom codebases to the new requirements by adding or removing custom functionalities – eventually introducing new OEM-specific private APIs and removing unused ones. From a security standpoint, removing unused private APIs, which we name *Residuals*, is highly important. Unused functionality not only increases code complexity but also broadens the attack surface. Many serious software vulnerabilities in commodity software and platforms are rooted in features that are never used [1].

Several research efforts [28, 35] have been proposed to investigate the phenomena of unneeded API removal from Android codebases, including deprecation practices, developer reactions and compatibility aftermath. However, to the best of our knowledge, no effort has looked into the security implications of failing to remove them. In this paper, we bridge the gap by performing a large-scale security investigation of Residual APIs. Our study aims to answer whether Android Residuals *do unnecessarily open the door to security flaws* as in other software and platforms.

To conduct the study, we put forward a solution that detects Residuals and evaluates their access control enforcement within custom ROMs. Our tool entails extensive program analysis of a large corpus of custom APIs (26,883), defined over our collection of 628 ROMs. Intuitively, a Residual API can be defined as any private API that is not used on a particular device but is used in earlier versions and/or in other models. This definition oversimplifies the nature of Residuals in Android. A seemingly unused API may be indirectly called through complex call chains and reachable through multiple framework entry points. To ensure accurate Residual detection, our analysis attempts to recover framework entry points through a specialized backward search over the framework classes.

The above definition further implies that the mere occurrence of unused APIs in a few isolated, random ROMs – without accounting for the APIs' *historical* and *model-specific* use patterns – may not accurately signal a Residual's presence. Our approach addresses this issue by building a usage history of custom APIs over our curated ROM samples. Specifically, Historical Residuals are detected by observing gradually or abruptly retiring APIs over time, while Model Residuals are identified by looking for specific use within clusters of devices from the same model or series.

To understand the security risks a Residual may pose, we perform a thorough security analysis. Our proposed analysis focuses on evaluating access control enforcement adopted by Residuals. The evaluation is guided by the intuition that, through various releases, Android APIs naturally evolve to add, fix and modify existing access control to patch vulnerabilities or add additional security requirements. Any failure to keep up with access control evolution will inevitably introduce anomalies and potential vulnerabilities. On the one hand, failing to adapt to the unstable device-specific implications of Android security features (e.g., permissions) will inevitably introduce security flaws. On the other hand, failing to keep up with the ever-evolving Android access control mechanism will lead to the adoption of obsolete security enforcement – thus unnecessarily re-opening the door to older vulnerabilities and invalidating current security requirements. Our proposed solution evaluates Residuals by inspecting implemented access control enforcement and verifying that it adopts *sound* security features and reflects *up-to-date* security requirements.

Our findings. Our study is the first to investigate Android Residuals and show that they are indeed pervasive. We run our tool to automatically evaluate 628 Samsung, LG, Blu, Xiaomi, Huawei, Lenovo and Asus ROMs, covering 7 OS versions and 105 unique device models. Our measurement study shows that Residuals are prevalent among all vendors, comprising up to 42% of the private APIs in a few models. While some are removed eventually, many Residuals are spotted even on the latest OS versions and device models across different OEMs, indicating that this security risk has yet to come to OEMs' full attention.

More importantly, our evaluation unveils that Residuals indeed carry evolution-related access control anomalies; including using undefined custom permissions, relying on nonexistent package names for security checks and even enforcing obsolete access control. Overall, our analysis identified 23% of the reported Residuals to be weakly protected. To estimate the risk magnitude of weakly-protected Residuals, we selected and investigated a set of reported instances (93), for which we had a physical device. We were able to exploit 8 different instances to conduct high-profile attacks. We found that a Residual on Samsung's S10 models can be used to intercept user key and screen taps without any permission requirements, thus allowing any third-party app to develop a powerful keyloggger stealthily. We exploited another two Residuals on Samsung's A/J Core series and the LG Q6, caused by device-incompatible and unsound security features, to write two more keyloggers. We reported the high-profile Residuals to the corresponding OEMs; Samsung and LG have acknowledged 8 vulnerabilities. So far, 5 have been issued CVEs with a NIST rating ranging from 7.1 to 9.8. We note that for one of the reported cases, which allows us to inject and corrupt secure system providers, the vendor's proposed fix entails recommending the vulnerable Residual for removal. To validate the scope and magnitude of Residual security at large, we further evaluated the reported instances based on several metrics. Our findings point to *unnecessary* security risks imposed by Residuals and an urgent need to debloat them.

We summarize the contributions of the paper as follows:

- We perform the first systematic large-scale study of custom Residuals. Our study unveils the extent and prevalence of Residuals and, more importantly, demonstrates that Residuals do indeed open the door to various attack vectors.
- We develop a set of new analysis techniques that detects and evaluates the risks of Residuals. Our techniques are specially tailored to detect evolution-induced access control vulnerabilities.

2 BACKGROUND

In this section, we lay the necessary background on the evolution of private OEM APIs. We then elaborate on a possible security outcome to motivate our study.

2.1 Private APIs Evolution

As mentioned earlier, OEMs aggressively customize the AOSP baselines. For each new Android version and device model, OEM developers adapt their codebases to new functional requirements by adding, altering and removing APIs. This extensive API retrofitting process usually spans Android SDK APIs as well as OEM-specific private APIs. When retrofitting SDK APIs, OEMs must abide by Google's regulations to meet compatibility requirements. That is, APIs designated for use, deprecation and removal by Google should be similarly designated by OEMs.

However, when it comes to OEM private APIs, the process is less regulated. Private APIs, provided to support internal framework and preloaded app developers, are added and removed frequently (~880 and 92 times, respectively, as reported in our dataset). This under-regulation coupled with OEMs' efforts to provide a one-size-fits-all framework implementation contributes to the production of *bloated* custom codebases. OEM devices tend to include a substantial number of private APIs [11, 32] (reaching up to ~3,500 in Samsung versions 7.0.1 and 8.1), some of which do not even fit with the devices' functional requirements. We refer to such unused APIs as *Residuals*.

Residuals not only increase code complexity but also induce compatibility issues. For instance, invoking an unsupported API on a particular device will lead to app or system crashes. Even worse, when Residuals provide sensitive operations and are not properly protected, they *unnecessarily* induce security issues. This is particularly inevitable when OEMs fail to adapt *up-to-date* and *compatible* security checks to safeguard a Residual's functionality.

2.2 Safeguarding Residuals

To deal with compatibility issues and to properly protect a Residual's functionality, OEM developers implement safeguards. At a high level, the guards attempt to reduce the pool of devices on which a Residual may be activated or restrict the callers to a set of verified entities (the expected users of the Residual). Specifically, the safeguards fall into the following two categories:

```
1  boolean startATTEntitleforTethering(...){
2    if(SystemProperties.get("ro.build.characteristics").equals("tablet")){
3      if(!telephonyManager.hasIccCard()){
4        Log.d("WifiService", "tablet has no sim card");
5        return false;
6      }
7    if(SystemProperties.get("ro.build.target_country").equals("US")
8      &&  SystemProperties.get("ro.build.target_operator").equals("ATT"))
9        if(getAppName(Binder.getCallingUid()).equals("com.smartcom"))
10         // perform actual functionality
```

Figure 1: Configuration Checks in a Custom LG API

(1) Configuration Checks: These guards are adopted to ensure that an API's provided functionality is compatible with the current release and/or is supported on the running platform. For example, for legacy APIs targeting obsolete functionalities, the guards make sure that the functionality cannot be triggered in newer releases. Similarly, for APIs supporting specific capabilities, the configuration guards ensure that the running platform embeds the corresponding hardware. Figure 1 depicts a few configuration checks implemented by LG within a custom API assisting AT&T tethering. Lines 2-3 ensure that the API can only be triggered on devices with mobile data capabilities; that is, if the device is a tablet, it should embed a SIM card. Other checks at lines 7 and 8 verify that the functionality can only be triggered in devices operated by the US-based carrier AT&T. Even if the API is introduced on devices not conforming to these checks, its functionality is safeguarded.

(2) Access-Control Checks: These checks reflect traditional Android access control enforcement. In this scenario, they restrict access based on unforgeable properties (e.g., UID) or acquired permissions. The calling entities reflect system processes or preloaded apps that exist on the devices where the Residuals are active.

While certain access control checks are intrinsically sufficient to properly protect a Residual, we observe that other checks implicitly rely on a co-located configuration check for validity. Without this secondary configuration check, the access control may be totally flawed. Consider the check performed at line 9 of Figure 1, which verifies that the calling app matches the name *"com.smartcom."* Observe that this check is *unsound* by itself since a package name can be squatted. Unless the package exists on the device, any third-party app can claim to be *"com.smartcom"* and trigger the privileged functionality. In this case, we found that *"com.smartcom"* comes preloaded on AT&T models, implying that the package check is actually sufficient under AT&T builds. Hence, the configuration check at line 8 validates the soundness of the package check.

Given these intrinsic complex properties, coupled with the prevalence of Residuals and the fast-paced Android updates, we argue that ensuring proper and valid safeguards is challenging and error prone. Access control vulnerabilities may be *unnecessarily* introduced because of Residuals.

3 PROBLEM

Since Residuals are deemed unnecessary and, at times, not intended for deployment on a particular device, framework developers may naturally overlook their implementation during integration and version upgrades. *Evolution-induced access control errors* are particularly dangerous in Residuals. On the one hand, a failure to account for the unstable device-specific implications of security features will inevitably introduce security flaws. On the other hand, a failure to keep up with the ever-evolving Android access control mechanism will lead to the adoption of obsolete security enforcement – thus unnecessarily re-opening the door to older vulnerabilities and invalidating current security requirements. Our study reveals a plethora of vulnerabilities resulting from these failures, including enabling third-party apps to exploit a Residual to access sensitive resources (such as the input driver).

In this paper, we aim to investigate the problem at large. Specifically, we seek to answer the following critical questions:

- RQ1: Do vendor developers adopt *sound*, *device-compatible* security features while enforcing access control in Residuals?
- RQ2: Do vendor developers propagate *up-to-date*, *consistent* access control enforcement in Residuals throughout version upgrades?

3.1 Unsound Security Features

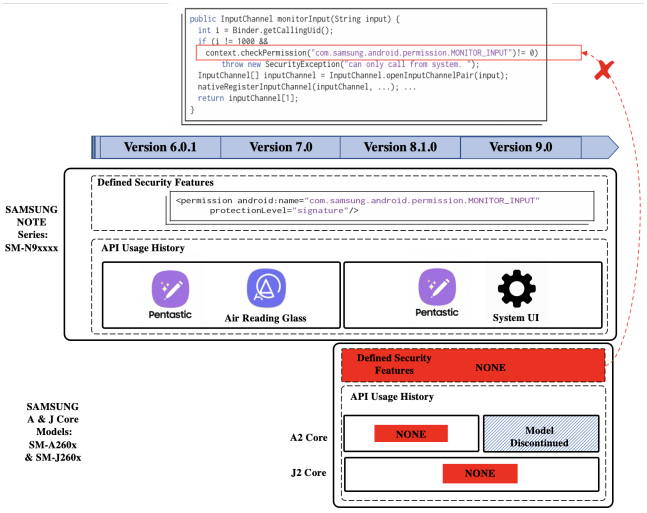

Figure 2: Usage of Undefined Permissions in Residual APIs

The correctness of access control enforcement heavily relies on the soundness of adopted security features (e.g., permission, calling uid, package name). While some security features are persistently sound (e.g., relying on the calling UID to verify that the caller is SYSTEM), others may imply different protections depending on the running device and model. Hence, if OEM developers do not account for these changes, a Residual API may use *incompatible* and *unsound* features that imply protections only available in other devices where the Residual is active.

Motivating Example. Consider the case depicted in Figure 2. As listed at the top, Samsung introduces a custom API `InputManager.monitorInput(...)` in a few device models. The API creates an input channel that receives input events from the input dispatcher. It can thus be used to intercept and monitor input events such as screen tap coordinates and key presses. Given the sensitivity

of the operation, Samsung enforces high-privilege requirements. The caller must belong to the system process (enforced through the check `getCallingUID() = 1000`) or hold Samsung's custom permission `com.samsung.android.permission.MONITOR_INPUT`. Thus, a third-party app cannot invoke this API unless it can somehow obtain the permission.

The lower parts of Figure 2 depict the API's related security definitions and usage history in Samsung Note Series and A/J Core Series. As illustrated, in the Note Series, the API is used in versions 6.0.1 to 7.0 (from Oct'15 to Oct'16) by two preloaded apps: *Pentastic* and *Air Reading Glass*. Starting from versions 8.1 to 9 (from Dec'17 to Sep'19), the API is used by *Pentastic* and another preloaded app *System UI*. Observe that the devices define the API's required permission `...MONITOR_INPUT` and designate it a *signature* level protection, which cannot be acquired by third-party apps.

In contrast, consider the usage history of the API in the Samsung A/J Core Series, illustrated at the bottom. As shown, the API is introduced in the first release of the devices (version 8.1, Dec'17) and has been consistently defined up to version 9 (Aug'18) in J2 Core (note that A2 Core was discontinued). However, no active usage site has ever been identified throughout the versions – thus making the API a Residual in A/J models. Though the API seemingly enforces access control checks, it is actually vulnerable. Since the API is deemed nonfunctional, the framework developers have overlooked defining the required permission (i.e., `...MONITOR_INPUT`).

Since the permission is undefined, any entity that defines it can acquire it and subsequently trigger the Residual's privileged operation. In this particular case, we were able to exploit the Residual to develop a keylogger without any permission requirements. Observe that the vulnerability has been dormant since its introduction in Dec. 2017, in part because the API *has never been used* since then. The issue has been acknowledged and fixed by Samsung. [1]

We note that undefined security features may also occur in non-Residual APIs. However, as uncovered by our study, they are substantially more prevalent in Residuals (refer to Section 8.3).

3.2 Obsolete Access Control Enforcement

Android has expanded beyond the traditional smartphone to support other device types and use scenarios. Along with the expansion, new security features and requirements are incrementally added with each update. A failure to keep Residuals up-to-date and compliant with the new requirements can cause anomalies.

Motivating example. Figure 3 depicts the access control evolution of two APIs: AOSP's `getDeviceId()` and LG's `getDeviceIdForVZW()`, both allowing the caller to read the device's ID (e.g., IMEI). We note that LG's API is defined in different models (versions 5-8), but is only used in VZW-specific models. Thus, it is a Residual in all other models.

As shown, AOSP's access control has evolved from enforcing a single *dangerous* permission `READ_PHONE_STATE` in versions 5.0-5.1.1 to requiring two different protections in versions 6.0 to 8.1 – either the permission `READ_PRIVILEGED_PHONE_STATE` or the permission `READ_PHONE_STATE` as well as explicit user approval indicated by an `AppOps` operation check. In contrast, LG's Residual has not seen a similar update, instead still adopting the obsolete single permission requirement. Under this anomaly, a malicious app could exploit the weakly protected Residual to read the device's IMEI. We have confirmed the vulnerability and reported it to LG. [2]

It is worth mentioning that starting from Android 10, Google prohibits third-party apps from accessing non-resettable identifiers such as the IMEI. AOSP's `getDeviceId` returns `NULL` in devices running 28 and older. Yet LG's Residual still returns a valid id.

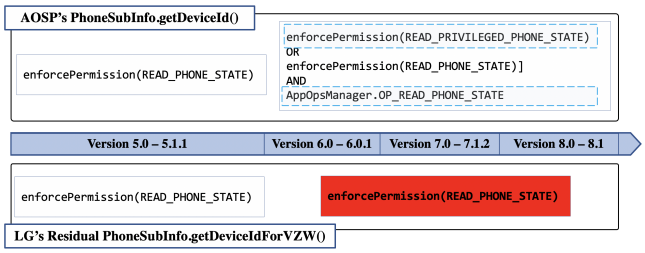

Figure 3: Access Control Evolution of Two LG APIs

3.3 Our Solution

In this paper, we investigate the prevalence and magnitude of access control anomalies in Residuals at a large scale. We analyze a broad spectrum of custom factory ROMs manufactured by 7 vendors, tailored for 105 models and spanning 7 versions. Our proposed investigation proceeds as follows. First, through program analysis of framework and preloaded apps, we recognize and pinpoint potential Residual instances in a ROM. We then build and investigate their usage patterns over a set of curated ROM samples. Confirmed Residuals (e.g., those following declining, retiring usage trends) are then fed to our proposed security analysis. We statically analyze the confirmed instances to identify the presence of unsound security features and obsolete access control checks.

The overall accuracy of the system relies heavily on the correct identification of Residual APIs. In light of custom call chains and API use patterns, detecting APIs that are defined but not used is not straightforward. Specifically, the detection entails the following two challenges:

Challenge 1: Identifying Entry Points Leading to a Target API. We identify through our analysis that custom APIs are often not directly invoked by other preloaded apps and framework services (hereafter referred to as components). Rather, they are usually wrapped in *Manager* APIs that are transitively wrapped around framework methods from both OEMs and Android; hence forming a long call chain from the components to custom APIs.

Consider Samsung's custom API `IUrspManager.setUrspBlackListUidRule(...)` shown in Figure 4.

The API is introduced in most SM-G38xxx models. As shown, it is wrapped in a custom Manager API `UrspManager.setUrspBlackListUidRule()`, which is transitively called by four other methods. The call chain is depicted by the dashed arrows. First, it is invoked directly by `disableMdo()`, a custom method added by Samsung to AOSP's `ConnectivityManager` class. The `disableMdo()` method

[1] CVE to be issued as soon as reported by Samsung.

[2] The issue has been acknowledged by LG.

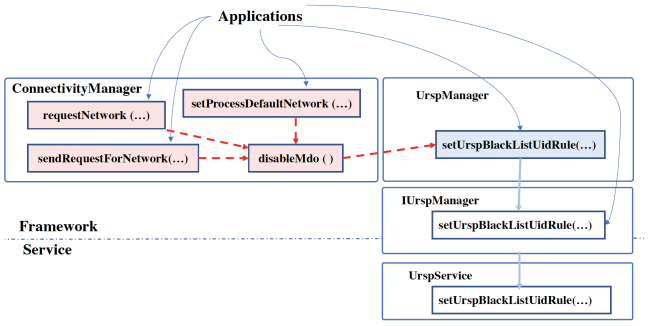

Figure 4: Multiple framework entry points leading to the custom API `IUrspManager.setUrspBlackListUidRule(...)`

is in turn called by three other methods within the same class. All together, the call chain introduces five valid framework entry points to the target `IUrspManager.setUrspBlackListUidRule(...)`. Apps can call any of them to trigger the target. To correctly detect Residuals, our analysis recovers all entry points for each API instance through a specialized backward search over the framework classes. More details are in Section 5.1.

This technique guarantees that we do not miss a target's active usage points within the device. It also avoids wrongly flagging certain APIs as Residuals even if we cannot identify an active site. Specifically, observe in the above example that `ConnectivityManager.requestNetwork(...)` is a public Android SDK API, implying that the private API `IUrspManager.setUrspBlackListUidRule(...)` is designated by Samsung to be indirectly reachable to third-party apps. Obviously, even if our analysis does not spot a usage point, the API could still be invoked via the public SDK API by other third-party apps to be installed later. We leverage this observation to rule out inspecting custom APIs reachable through public SDK APIs from our analysis. Specifically, after recovering framework entry points for a target API, our analysis proceeds to identify its usage points only if it is not transitively reachable via a public SDK entry point. Thus, `IUrspManager.setUrspBlackListUidRule(...)` will be skipped.

Challenge 2: Recognizing Residual Patterns over Time / Models. Studying one Residual instance within the whole population may not reveal interesting properties. As such, we must clearly define our investigation scope to infer meaningful Residual access control properties. To this end, we formally group Residuals into two categories based on their usage patterns: (1) *Historical Residuals* denote APIs that were active in *older* models but have ceased being used in successor and new models while (2) *Model Residuals* reflect APIs that are exclusively active on select device models from various versions. We detect Historical Residuals by observing the usage history of the APIs and recognizing the ones retiring (gradually or abruptly) over time. In contrast, to detect Model Residuals, we cluster similar devices (at the series or model level) and identify model-specific usages regardless of the version.

In the next section, we describe our solution in detail and elaborate on how we solve the above challenges.

4 OVERVIEW

To investigate Residuals at large scale, we design and implement ReM [3], a set of new analysis techniques that detect and evaluate the risks of custom Residuals. In this section, we first present our high-level idea and then describe the details of the proposed techniques.

Architecture. ReM is composed of three components: a ROM Analyzer, Usage-Pattern Extractor and Risk Identifier. Given a set of custom APIs in a device, the ROM Analyzer identifies *Likely* Residuals through a synergy between framework and preloaded app analysis. At the framework layer, ReM exhaustively collects public entry points that transitively lead to the invocation of the custom APIs. Through preloaded app analysis, ReM identifies *live* usage sites leading to a custom API either directly by calling the API's Remote Procedure Call (RPC) point or indirectly by calling the identified entry points. Unused APIs are flagged as *Likely* Residuals.

The Usage-Pattern Extractor confirms *Actual* Residuals through a large-scale cross-ROM analysis. Specifically, the module identifies *Historical* Residuals by running the above analysis repeatedly over a pool of curated ROMs for a target vendor, ordered by release date. It similarly recognizes *Model* Residuals by running the analysis over a cluster of ROMs from the same model/series. It subsequently builds a usage history for each *Likely* Residual in an attempt to identify the ones conforming to *Actual* Residual patterns characterized by an abrupt or a gradual retirement over time, or reflecting a consistent model-specific use.

Finally, Residuals are handed over to the Risk Identifier, which performs specialized program analysis on the Residual implementations to uncover two potential flaws: (1) Unsound security feature use and (2) obsolete access control. To detect the former, it leverages a set of patterns indicating unsound feature use and looks statically for their presence. Examples of these patterns include the use of package checks without co-located configuration checks and the use of an undefined permission. To detect the latter, the Risk Identifier performs a highly-optimized inconsistency detection and accordingly infers anomalous obsolete access control enforcement. It finally reports vulnerable Residuals.

5 AUTOMATED DETECTION OF RESIDUALS IN CUSTOM ROMS

Given the sheer number of analysis targets (framework and system app classes) and the large number of ROMs required for the historical analysis, ReM 's analysis must be scalable and efficient.

5.1 Identifying Likely Residuals in a ROM

ReM conducts program analysis of the framework and preloaded apps to detect unused custom APIs. It first identifies framework-level entry points leading to the custom APIs and then statically looks for usage sites leading to the invocation of the APIs or corresponding entry points.

As mentioned earlier, identifying framework entry points is important since OEM private APIs are often available to framework and system app developers through custom *Manager* APIs (e.g., `UrspManager.setUrspBlackListUidRule` in Figure 4). These *Manager* APIs may be transitively invoked by other internal framework

[3] ReM: Short for *REMNANT*

and SDK methods, forming indirect call chains from the components to the custom APIs.

Collecting framework entry points: We first use Wala to process the framework libraries and extract defined classes and methods. Now, performing a forward search on each method to extract reachable APIs may sound compelling. However, it is likely that it will encounter and analyze many irrelevant methods and code fragments, unarguably affecting the scalability of the overall detection. To tackle the issue, we propose a more focused approach. We start with our set of target custom APIs, and perform backward expansion to iteratively discover public calling methods. Specifically, we use WALA to perform a class hierarchy analysis of the extracted classes and methods. Then for each method, we perform a depth-first reachability analysis on its call graph and locate the occurrence of a target custom API. If the latter is located, the calling method is added to the set of the target API's callers and is transitively fed back to the analysis loop to locate its potential public callers. The backward exploration constructs a mapping between each custom API and its calling methods and stops once no public callers can be encountered. Since the call chains are inherently deep, we optimize the exploration by:

- Caching discovered caller-callee mappings. The exploration consults the cache before moving on to look for other callers in order to avoid duplicate path exploration.
- The exploration stops preemptively if a public SDK method is encountered. That is, if a caller matches the name of a public API (which we have compiled for each Android release), the target API is ruled out from further analysis since it can be invoked by third-party apps. We further rule out the public API's direct and transitive callees from subsequent analysis, essentially considering the whole call chain accessible to third-party apps.

Collecting usage points in apps: In this task, we statically analyze the apps and internal framework classes to collect usage points of a target API. Specifically, for each app, we perform standard forward reachability analysis starting from the app's public entry points (Android component life cycle methods and callback methods) and search for invocations to the targets. The analysis looks for invocations to the API's exposed Binder method and to its extracted framework entry points. To optimize the exploration, the search prioritizes entry points at the top level of the recovered caller-callee mappings chain and skips looking for a callee if a caller has already been encountered. Our analysis further handles calls to the APIs through Java reflection. During the reachability analysis, we treat reflection call methods as potential sinks if the arguments match the API's recovered framework entries or the RPC method itself. Specifically, for each Java reflection call that allows method invocation, we perform string analysis to extract the value of the call parameters (class names and method names). We use constant propagation within an analyzed app's inter-procedural CFG to resolve the method name in a reflective call (e.g., method.invoke(object)) and the class name that the method belongs to. A string variable from external input is modeled by a special value that denotes any string. We note that we are not interested in resolving the type/values of the arguments passed. This is sufficient for most of the cases we encountered.

Collecting usage points in system services. We further look for call sites to the target APIs in the system services classes. We note that triggering the system service functionality may be initiated by the system server itself (e.g., in init methods, inner methods not exposed through IPC, etc.) through non-traditional channels (e.g., from the native layer). Thus, we mark any API that is triggered on the server side as a used API. Observe that this approach is conservative and is likely to overestimate the usage sites of APIs since a recovered site might not be necessarily invoked (i.e., it might occur in a dead code area).

At this stage, identifying *Likely* Residuals is straightforward; unused custom APIs are flagged as *Likely* Residuals.

5.2 Characterizing and Confirming Residuals

As stated earlier, we categorize Residuals based on their usage patterns, as follows:

(1) A likely Residual is a Historical Residual if it gradually or abruptly retires over time. That is, the API's usage pattern decreases over time, until it is no longer in use in new successor devices.
(2) A likely Residual is a Model Residual if it is consistently used in specific device series and models but not in others.

Observe that the two categories are inherently overlapping, since Model Residuals may also become unused over time.

ROM collection and curation. To detect Historical and Model Residuals, we perform a broad analysis of 628 custom ROMs released over the last ~10 years (from Oct 2011 to May 2021). These ROMs are representative of major mobile vendors. More details on the sample ROMs can be found in Section 7.

We curate the samples for our analysis by carefully considering the following three properties of a ROM: (1) vendor, (2) model and (3) release date. We construct a usage history for a given API by analyzing chronologically ordered ROMs produced by the same vendor. We similarly build model-specific usage by grouping ROMs from similar series and models.

To identify the properties, we process a ROM's `build.prop` file (containing device properties) and extract the values of `ro.product.brand`, `ro.product.model` and `ro.build.version.release`. Note that a few vendors customize these attributes so we had to treat them on a case-by-case basis.

Scope of analysis. ReM builds the usage patterns of the likely Residuals by running a per-ROM analysis over our curated pools of samples. In total, our analysis involved inspecting 48,000 unique preloaded apps (more than 250,000 all together) and led to identifying 6,349 custom APIs that exhibit actual Residuals patterns. More details can be found in Section 7.2.

In the next section, we describe how we evaluate the detected Residuals' security properties.

6 AUTOMATED SECURITY EVALUATION OF CUSTOM RESIDUALS

In this section, we evaluate Residual access control enforcement. Our focus is on *evolution-induced access control vulnerabilities* that arise when framework developers do not safeguard Residuals. We classify these vulnerabilities as either unsound security features or

obsolete access control enforcement. Unsound security features include undefined and device-incompatible features. Obsolete access control occurs when Residuals are not maintained and their access control enforcement is not updated and strengthened along with non-Residual APIs.

Evaluation scope. We note that both classes of evolution-induced access control vulnerabilities examined in our security evaluation result from the presence of unused functionality. We focus on these particular vulnerabilities since, intuitively: (1) unused functionality is likely to be overlooked during updates and model customization and (2) in many cases, unused functionality is not even intended for use on a target device. Other types of vulnerabilities – particularly those that are *equally likely to occur in used APIs*, such as improper input validations, are out of scope for our evaluation.

Next, we describe how ReM detects the two classes of evolution-induced access control vulnerabilities.

6.1 Unsound Security Features

As stated earlier, the correctness of access control enforcement heavily relies on the soundness of adopted security features. Certain features may imply different protections depending on the running device version and build characteristics. Thus, a sound feature on a device where an API is used might not be sound on other devices where the API is not used.

Undefined custom permissions and broadcasts. Custom permissions and protected broadcasts are introduced by customization stakeholders to protect custom resources. They are added, removed and renamed frequently. Removing a custom permission is performed when the defining stakeholder is not involved in a particular customization or when the permission is not needed. Other custom permissions are introduced by vendors and are tightly related to hardware. They are debloated when the corresponding resource is considered *nonfunctional*. For example, Samsung may remove permissions required to access its `Pen` functionality if the device does not embed a physical pen hardware. Protected broadcast definitions are removed for similar reasons.

Removing custom permissions and protected broadcast definitions is largely fine when *all APIs* referencing them are simultaneously removed. However, in the case of unmaintained Residuals, the occurrence of such references is highly problematic. Using an undefined security feature is unsound. As reported by the study [13], any app that defines removed features can silently gain the privilege to access the components referencing them. (Refer to Section 3 for an example.) To detect this pattern, ReM performs the following analysis:

- For each reported Residual, ReM statically extracts its implemented access control enforcement and identifies used security features. Specifically, it first builds the Residual's interprocedural Control Flow Graph (CFG) and traverses it to extract invocations to security-relevant APIs (e.g., `checkPermission`, `enforceCallingPermission`). It then traces back from the APIs and keeps track of the permission string constants passed as arguments. ReM similarly processes registration sites of framework-defined broadcast receivers to extract corresponding actions.
- For each ROM with Residual instances, ReM collects the definitions of security features by running an XML parser over the framework and preloaded apps' manifest files.
- Last, ReM conducts a differential analysis to pinpoint Residuals that use undefined security features.

Package name checks without a co-located check. Using package names for access control enforcement is not always sound. Since the names are forgeable identifiers, any party can squat the property and pretend to be the caller. In the Residuals scenario, since the expected calling package does not exist, the property is forgeable. Nonetheless, the property may become sound if used in conjunction with other checks. As stated earlier, configuration checks can validate a package name check. Traditional checks such as signature checks and other persistently sound checks (e.g., UID checks) naturally strengthen package name checks.

To detect the use of unsound package name checks, ReM traverses a target Residual's interprocedural CFG to collect invocations to the following: (1) APIs that retrieve the package name of the caller (e.g., `PackageManager.getNameforUid` and `PackageManager.getPackagesForUid`), (2) signature checks (e.g., `PackageManager.checkSignature`) and other checks for extracting the caller's unforgeable identifiers and (3) configuration checks. ReM then inspects the collections and marks sole invocations to package name checks as *potentially unsound*. Last, ReM verifies whether the target ROM does not include the specified package name to confirm unsoundness.

Resolving strings. We observe through our analysis that package names returned from the `PackageManager` APIs (e.g., `getNameForUid`) are sometimes compared with dynamically constructed strings; i.e., by concatenating substrings, including constants, parameters and return values of other methods. We employ def-use analysis and examine if the package name returned from the target `PackageManager` APIs is compared with a string. We then use interprocedural backward slicing and forward constant propagation to transitively resolve the strings. String arguments to other package check APIs (e.g., `PackageManager.getPackageUid`) may also be dynamically constructed and we resolve them similarly. We model strings that cannot be statically resolved (e.g., read from a framework resource file) with a placeholder that denotes any string. Our analysis conservatively considers a package name string that cannot be fully resolved to be sound.

Collecting custom configuration checks. Besides using common AOSP APIs (e.g., `SystemProperties.get()` and global static fields (e.g., `OS.Build`), we observe through our analysis that vendors use a variety of custom methods for device configuration checks. Our inspection shows that these methods are often wrappers around AOSP APIs and usually involve multiple call chains. While performing inter-procedural CFG traversal will ultimately discover the underlying AOSP checks, it will encounter many irrelevant methods and affect the overall extraction performance. We tackle the issue by performing a one-time per vendor backward propagation (similar to the approach discussed in Section 5.1). The backward exploration builds a mapping between AOSP configuration check APIs and their calling methods, which we manually inspect to filter out custom configuration checks.

For each vendor, the automated backward propagation yielded 42 to 74 candidate configuration check methods. Our manual filtering

yielded 19 to 24 actual configuration methods per vendor. We note that the manual filtering process is a small scale, one time effort.

References to deprecated security features. For graceful removal of a security feature, framework developers may first flag it as deprecated, through the Java *@Deprecated* annotation. The deprecation subsequently pressures the developers to refactor their code and migrate to other alternative features. Eventually, after a few releases, the deprecated features are removed.

While the use of a deprecated security feature is not a vulnerability per se, we argue that it may eventually lead to one. Since Residuals are not used, they may not be properly maintained throughout version upgrades, leading to the persistence of deprecated security feature usage, even after the feature removal.

To detect this pattern, we use Wala to extract the Java annotations associated with the definition points of permissions and protected broadcasts (defined in the class `Manifest$permission`) and flag those annotated with `java.lang.Deprecated`.

6.2 Obsolete Access Control Enforcement

Android APIs are continuously evolving to add, fix and modify enforced access control. The evolution addresses new security requirements (e.g., migrating from a single-user to a multi-user device) and fixes reported flaws. A failure to keep up with the fast-paced evolution could induce obsolete access control enforcement, which may reflect weaker or absent access control enforcement.

Recognizing obsolete access control enforcement is not straightforward. Residuals implement custom functionality, with no publicly-available security specifications. As such, it is challenging to infer whether enforced access control is up-to-date. A popular approximate solution is to perform consistency analysis – essentially, comparing the access control enforced across multiple paths to the same resource and reporting inconsistencies; i.e., one path includes access control while the other does not. Various work exists in the area, ranging from approximate solutions [14, 16, 32] to more precise ones [11]. ReM follows an adapted version of the former approach since conducting a path-sensitive analysis will not scale to tackle the sheer number of APIs in our studied ROMs.

ReM conducts a largely-localized convergence analysis to identify other framework APIs that converge in functionality with the reported Residuals. It then extracts access control enforcement along the new APIs and compares them to those enforced by the Residuals. Observe that performing a framework-wide convergence analysis would not scale as some ROMs are extensively customized (e.g., more than 2000 custom APIs). To speed up the analysis, we limit our convergence analysis to (1) APIs defined within the same system service and (2) APIs defined in system services providing similar functionality. We leverage similar naming patterns to infer whether two system services provide overlapping functionality (e.g., *SemClipboard* and *Clipboard* services, *ISmsEx* and *ISMS* services).

To infer whether a Residual reflects updated access control, ReM further conducts a cross-ROM inconsistency analysis similar to [11]. Specifically, ReM compares the access control enforced by an API across multiple ROMs with different use scenarios; i.e., cases where the API is used in one but the Residual in the other.

We applied ReM to evaluate the access control enforcement of the Residuals identified in our ROM samples (i.e., 6,349 Residuals). ReM uncovered 1,453 violations. Details about the Residuals landscape and pertaining security properties are discussed next.

7 LARGE-SCALE MEASUREMENT STUDY

Table 1: Collected ROMs

EOM	Statistics (#)	API Level / Version Numbers						
		19 4.4 - 4.4.4	**21-22 5.0 - 5.1.1**	**23 6.0 – 6.0.1**	**24-25 7.0 - 7.1.2**	**26-27 8.0 - 8.1**	**28 9**	**29 10**
Samsung	ROMs	16	23	61	48	52	116	49
	Models	13	22	29	33	15	32	17
	APIs	3482	2462	4273	3588	3454	2282	2386
	Apps	168	257	308	310	315	335	345
Blu	ROMs	16	14	31	14	3	2	2
	Models	12	11	26	11	3	2	1
	APIs	403	516	582	562	636	476	794
	Apps	107	108	99	109	97	122	101
LG	ROMs	6	7	5	4	3	4	4
	Models	3	3	4	3	2	3	3
	APIs	1352	1017	875	1422	1101	902	896
	Apps	140	151	104	159	214	202	237
Xiaomi	ROMs	3	2	4	2	2	5	3
	Models	3	2	4	2	2	4	3
	APIs	773	962	1033	714	771	589	539
	Apps	133	154	181	182	182	207	197
Huawei	ROMs	17	1	2	2	2	3	7
	Models	13	1	2	2	2	1	1
	APIs	1233	157	963	576	725	461	286
	Apps	109	115	119	89	97	93	143
Lenovo	ROMs	38	11	6	2	2	4	1
	Models	31	8	4	2	2	3	1
	APIs	1651	1375	990	820	556	373	199
	Apps	121	131	135	113	156	143	111
Asus	ROMs	3	3	2	5	2	2	2
	Models	2	2	1	3	2	2	2
	APIs	1064	773	948	892	599	364	89
	Apps	147	167	131	161	179	176	156

To measure the pervasiveness of Residuals in the fragmented Android ecosystem and to understand the scope and magnitude of access-control anomalies they may pose, we perform a large-scale study of 628 ROMs.

Study Setup. The study has been conducted using 4 server machines equipped with 1/4 TB RAM, 16 cores, 64 Gbps net, 4 NVIDIA K10 GPU cards, each containing 2 GK104 GPUs.

7.1 Data Collection and Processing

Factory ROMs collection. We collected 628 custom ROMs released over the last ~10 years (from Oct 2011 to May 2021). The ROMs cover 7 major releases (from 4.0 to 10) and are customized by 7 vendors and cover 105 device models. We developed a crawler that automatically downloads vendor ROMs from public repositories. The crawler tries to cover as many distinct models and versions as possible to identify Historical and Model Residuals.

Table 1 lists the detailed statistics of our collected dataset. As shown, the ROMs are representative of big and medium players in the mobile market. We note that unlike Samsung and Blu ROMs, for which many dedicated public repositories are available, some vendor ROMs are more difficult to obtain and thus constitute smaller sample sizes in the dataset.

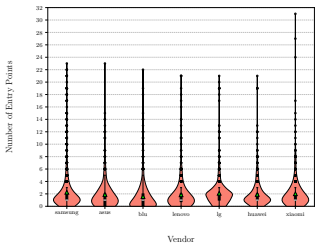

Figure 5: Distribution of the # of Entry Points per vendor

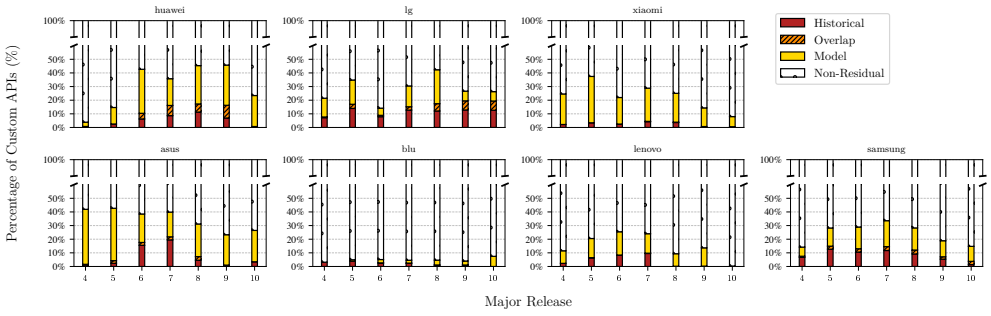

Figure 6: Residuals Breakdown

Framework, preloaded apps and configuration files Extraction. We preprocess the ROMs to extract framework classes and preloaded apps. To do this, we first locate the system partition, which contains the relevant classes.

Locating the System Partition. For Android versions 4 through 9, we search for the system partition within `system.img` or `system.img.ext4` files. For LG images, the firmware is often packaged in a `kdz` file. To extract the system partition from `kdz` files, a modified version of the `SALT` tool [8] is used to generate `dz` files and to extract the embedded system partition (`system.image`). Once the system partition has been located, `imjtool`[5] is used to extract the image file, which is then mounted. When working with Android 10 ROMs, the system partition identification process differs slightly. Android 10 introduces the dynamic partitioning system, which allows partitions to be resized, created and removed during over-the-air updates [4]. As a dynamic partition, the system partition is housed within a larger super partition (`super.img`). To unpack the super partition, the sparse image is first converted to a raw image using `simg2img` [9]. Then we unpack the raw image using the `lpunpack`[6] tool, obtain a `system.img` file and proceed to mount the system partition.

Extraction of Framework classes and Preloaded Apps. We extract each mounted image's build properties, specified in `build.prop` files. We then extract any framework or app `odex`, `vdex`, and `apk` files and use the `vdexextractor`[10], `baksmali`[3], `smali`[3], `apktool`[2], and `oat2dex`[7] tools to generate `dex` and manifest files.

7.2 Analysis Complexity

Codebases. Our analysis investigated 26,883 *distinct* private APIs and 48,000 preloaded apps. Table 1 presents the detailed statistics. The third row of each OEM entry lists the average number of *private* APIs recovered by ReM (not including AOSP's exposed APIs). As shown, vendors introduce 880 APIs, and the extent of customization differs between OEMs – with Samsung and LG exhibiting significantly more private APIs. As further illustrated in the fourth row of each vendor, the number of preloaded apps increases between major releases: it is larger in the latest releases. Observe that the results are reported as averages; some vendor-specific models include less preloaded apps than others (e.g., Samsung A/J Core models include ~30% less apps compared to ZFlip models).

Recovered entry points. As discussed in Section 5, ReM collects framework entry points leading to custom APIs to accurately detect Residuals. Figure 5 reports the distribution of the recovered entry points, per OEM. For all OEMs, 50% of the APIs have 1 to 2 entry points; 25% have no entry points (meaning that the API is solely invoked via its RPC entry); and 25% exhibit a significantly larger number reaching up to 31 for Xiaomi. We investigated a few randomly selected samples that fall in the last category and found they often corresponded to methods for accessing custom information, e.g., custom profile information, whitelists for different services and keyguard information. Clearly, performing a simpler analysis that relies only on the RPC entry points and direct managers is likely to generate inaccurate Residual estimations.

7.3 Residuals Landscape

Among all the 26,883 extracted private APIs, ReM discovered 6,349 instances that are Residuals in specific models/series or at specific release versions. We reiterate that as per our Model and Historical Residual detection, a used API is only flagged as a Residual if it exhibits certain trends (refer to Section 3.3).

Figure 6 depicts a breakdown of the reported Residuals per OEM. As shown, Residuals are prevalent among all vendors, reaching up to 42% in LG and Huawei (major releases 8.0 and 9, respectively). Blu ROMs exhibit the lowest number of Residuals since they are the least customized (i.e., smaller number of private APIs). We further note that Lenovo records 2% Residuals in version 10 because it was the least customized out of all the Lenovo samples.

Observe that the number of Historical Residuals is lower in version 4, since the analysis cannot pick up the usage trend yet, as no data is available for earlier releases. The analysis only reflects the number of Residuals that are persistently unused on all releases.

Analysis Accuracy. From all the reported Residuals, we randomly sampled 50 and manually analyzed their usage in the corresponding ROMs (7 ROMs). We employed a simple word lookup to identify references to the Residuals (using the `grep` utility) in the preloaded apps and further investigated the references to verify if they were actively used (i.e., not included in dead code). We note that this analysis is simplistic since it is difficult to verify if a code region is dead, especially in the case of long call chains and obfuscated apps. Out of 50 instances, only 4(8%) were found to be false Residuals; that is, falsely reported to be not used while they were actually used in preloaded apps. Looking into these positives, we found that they occurred in obfuscated apps.

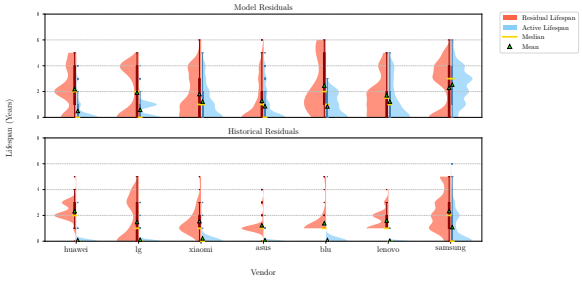

Figure 7: Violin Distribution of the Active and Residual Life Spans

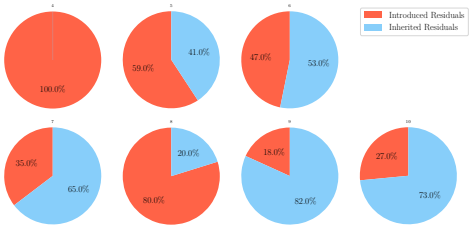

Figure 8: Inherited vs Introduced Residuals

To estimate missed Residuals, we have similarly sampled 70 reported non-Residuals and manually analyzed their usage in the corresponding ROMs (11 ROMs). In all 70 samples, 5 (7.14%) were missed by ReM ; i.e., Residuals considered to be used. We investigated these cases and found that they are caused by ReM 's reflection handling (see Section 5.1). Since we do not resolve the type/values of the arguments passed in Java reflective calls, ReM cannot distinguish overloaded methods. Other cases were, due to infeasible code paths, conservatively treated by ReM as feasible.

7.4 Residual Lifespans

Figure 7 displays violin plots representing the distributions of the active and Residual lifespans for each OEM's Residuals. The active lifespan is the total number of versions a Residual is being actively used by some framework service or preloaded app, while the Residual lifespan is the total number of versions from a Residual API's introduction to its complete disappearance. Problems arise when an active lifespan is shorter than its corresponding Residual lifespan, as is the case for most vendors depicted in Figure 7.

The density of each violin plot corresponds to the frequency that a given lifespan is present in the larger population of active lifespans or Residual lifespans. We can see that, consistently, in overlapping regions between Residual and active lifespan distributions, the density is much higher for the Residual lifespan distributions.

For both Model and Historical Residuals, we can further spot that the mean active lifespan is almost always lower than the mean Residual lifespan. Note that a mean lifespan of zero implies that our analysis, spanning versions 4-10, did not identify any ROM instance actively using the Residual.

7.5 New versus Inherited Residuals

As shown in Figure 6, the percentage of Residuals is higher in versions 7-8 and starts a downward trend in versions 9 and 10. Although this signals that vendors are debloating Residual APIs more notably in newer versions, the issue of Residuals is still prevalent among new versions. As shown in the Figure, Residuals exist in significant proportions in the latest versions; for example, LG, Huawei and Asus record between 23 and 28% Residuals in version 10.

To further demonstrate the importance of the Residuals issue in recent ROMs, we report the percentage of newly-introduced versus inherited Residuals throughout each new release. Figure 8 depicts the results; note that the results are aggregated for all vendors (per version). As shown, 27% of Residuals are newly-introduced; i.e., they were active in version 9. This experiment clearly demonstrates that Residuals are not an issue of the past. This observation has yet to come to the vendors' full attention.

8 RESIDUALS SECURITY LANDSCAPE

In this section, we answer the following research questions:

- RQ1: Do vendor developers adopt *sound*, *compatible* security features while enforcing access control checks in Residuals?
- RQ2: Do vendor developers propagate *up-to-date*, *consistent* access control enforcement in Residuals throughout version upgrades?

8.1 Unsound Security Features

Among all OEM Residual instances, ReM identified 978 flaws caused by the use of unsound security features. Observe that some of these flaws can be attributed to the same property (e.g., an OEM may use an undefined permission in three distinct Residuals, thus introducing three flaws). We also note that, although less common, multiple flaws may occur within the same API ($< \sim 3\%$).

Columns 2-7 in Table 2 depict a breakdown of unsound security features use per OEM. As listed, the number of flaws varies between vendors, with deprecated permissions being the least common in most vendors except for Samsung.

Undefined permissions are pervasive among Samsung samples. Examples include `com.samsung.accessory.manager permission.AUTHENTICATION_CONTROL`, `USE_LINK_TO_WINDOWS_REMOTE_APP_MODE` and `com.samsung.android.knox.permission.KNOX_EBILLING_NOMDM`, which ReM identified as causing more than 40 flaws in versions 9 and 10. All vendors used an unsound package check at the Residuals implementation. Examples include *com.sprint.**, *com.verizon** and **.docomo.**, which are left over from carrier-specific models. Lenovo and Huawei have the least flaws.

We note that the majority of our findings are spotted in Samsung largely because of its sample size (our collection includes more than 49 Samsung models as opposed to an average of 4 in other vendors).

8.2 Obsolete Access Control Enforcement

Columns 8-9 in Table 2 report the results of our conducted inconsistency analysis. As depicted, OEM Residuals do induce anomalies. ReM reported 14 to 442 inconsistency instances (505 all together), caused by the Residual leveraging a *different* security check to protect its underlying resources. We have inspected the results and

Table 2: Unsound Security Features Use

Vendor	Undefined Permissions		Deprecated Permission		Unsound Package Checks		Obsolete Access Control	
	Residual APIs	Used APIs	Residual APIs	Used APIs	Residual APIs	Used APIs	Residual APIs	Used APIs
Samsung	273	19	229	90	339	188	402	113
Blu	2	0	0	0	23	6	14	9
LG	6	0	2	1	29	13	102	37
Xiaomi	0	0	0	0	26	10	48	25
Asus	8	0	0	0	18	7	31	16
Lenovo	12	5	0	0	4	7	33	21
Huawei	0	0	0	4	7	11	21	15

confirmed that a significant proportion (~ 67%) exist due to OEMs overlooking the integration of *User* and *AppOps* checks. For example, LG adds 8 Historical Residuals in its custom *ISms* service which allows the handling of SMS functionalities without enforcing *AppOps* operation checks.

8.3 Comparison with Non-Residual APIs

Prevalence of Flaws among Non-Residuals. Evolution-induced anomalies may also occur in non-Residual APIs. Nonetheless, in contrast to Residuals, active APIs are better maintained and often undergo extensive security testing. To demonstrate that evolution induced flaws are less common in non-Residuals, we evaluate them using ReM . In Table 2, columns 3, 5 and 7 report the prevalence of unsound access control features and column 9 reports the number of detected inconsistencies. With the sole exception of Huawei's use of deprecated permissions in four non-Residual APIs, the flaws are significantly more prevalent in Residuals. In Section A of the Appendix, Figure 10 depicts a breakdown of the flaws. As shown, Residuals are responsible for most of the reported vulnerabilities.

Comparison of Access Control Updates. To demonstrate that vendors may overlook updating Residuals in comparison to active APIs, we perform another experiment. For each custom API, we approximate its received access-control related updates as follows: we build a history of its adopted access control enforcement over time and report the number of observed distinct checks. We then compare the estimated numbers for Residual and non-Residual instances. Figure 9 reports the results. Both Historical and Model Residuals tend to receive less updates than Active APIs.

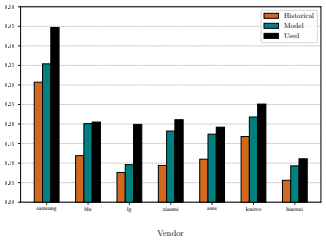

Figure 9: Average API Access Control Updates

9 EXPLOITING RESIDUALS

We note that *not every Residual is exploitable*. Clearly, just like any other Android API, a Residual is exploitable depending on its provided functionality (e.g., a Residual that provides less sensitive operations may not be exploitable). Nonetheless, a privileged Residual API can open the door for exploits. While the ideal fix for a Residual is through its removal, it can be protected by a *strong* access control requirement or by a persistent non-configurable device property. However, if the proper protections are not in place, a Residual can be exploited to achieve security damages.

9.1 End-to-end POCs

To understand the security issues Residuals may pose, we analyzed a small subset of the reported weakly protected instances (93 cases). Our selection of the targets was based on the following three criteria: (1) comprehensibility of the Residual code, i.e., we avoided instances referring to proprietary functionalities with no public description; (2) availability of physical devices (specifically, LG and Samsung) and (3) sensitivity of operations – we prioritized sensitive APIs. Our manual analysis confirmed 8 exploitable Residuals. A summary of the findings is presented in Table 3. Note that, though the exploits span different devices as reported by ReM , we are conservatively listing here only the devices on which the attacks were manually confirmed. We have reported our findings to LG and Samsung. 7/8 have been acknowledged and fixed. One instance was marked as duplicate. Next, we describe a few instances.

.

Injecting Data into Privileged Content Providers. Our historical analysis of the LG samples reveals another major vulnerable Residual. The victim API `ISms.insertDBForLGMessage(...)` is defined in all LG devices running 4.4.4 up to version 10 [4] but is only used up to version 8.0 – thus becoming a Residual in versions 9 and 10. The Security analysis module reveals that it enforces obsolete access control – it requires the permission `android.permission.RECEIVE_SMS` while another path enforces a System check. A further dive into the Residual's implementation reveals that it allows inserting data to *any Telephony-accessible* content providers, while solely enforcing the aforementioned permission. Specifically, the Residual takes as arguments a Uniform Resource Identifier (URI) along with content values and then inserts the supplied values into the URI. Since the defining service `ISms` runs within the content of the Telephony process, the Residual can be exploited to insert data to any privileged provider that the process has access to – e.g., `Settings.Secure` and `Settings.System` providers, which maintain secure/system preferences that apps can read but not write. We confirmed the vulnerability through a PoC that targets `Settings.Secure` content provider to automatically replace the default IME with our specified IME (e.g., containing malicious keylogging functionality). LG has acknowledged and fixed the vulnerability. It is worth noting that, as confirmed by LG, the fix for Android R entails *removing the API.*

Keylogger on LG. We have identified another Residual `IWindowManager.setInputFilter(.)`. on LG Q6 that exhibits a similar pattern to the previous example. The Residual is defined on a few LG ROMs from versions 4.4.4 to 10 but is only used up to version 8.0. The Residual allows intercepting and controlling all input

[4] The API may have been introduced before version 4.4.4.

Table 3: Confirmed Exploitable Residuals

Vendor	Model	Residual Location	Impact	Vendor Reaction	CVE	NIST Ranking*
Samsung	S9	InputMethodManager	Corrupt Service Manager Device Shutdown	Confirmed, Fixed	CVE-2018-21088**	High (7.5)
Samsung	S10	SPENGesture	Keylogger	Confirmed, Fixed	CVE-2019-20597	Critical (9.1)
Samsung	S9	PersonaManager	Alter OEM Lock configurations Disable Keyguard Features Alter Profile Restrictions	Confirmed, Fixed	CVE-2020-25055	Critical (9.8)
LG	LG Q6	WindowManager	Keylogger	Confirmed, Fixed	CVE-2020-12754	High (7.8)
LG	LG Q6	Isms	Insert data into system providers	Confirmed, Fixed	CVE-2021-30162	High (7.1)
Samsung	J2/ A2 Core	InputManager	Keylogger	Confirmed, Fixed	To be issued	–
Samsung	S6 Note	PersonaManager	Launch activities through the system	Confirmed, Duplicate	–	–
LG	LG Q Stylo 4	IPhoneSubInfo	Read phone IMEI	Confirmed, Not Fixing	–	–

*The severity metric is reported based on CVSS 3.x.
**We note that we re-discovered CVE-2018-21088 using our tool. The issue was initially discovered by us manually.

events before they are dispatched to the system or apps by registering an input filter. Alarmingly, our security module flagged the case as using an *unsound* security feature. Specifically, the API verifies if the calling package matches one of the two names: `"com.lge.systemserver"` or `"com.lge.onehandcontroller"`, and accordingly allows access to the filter registration. However, the API does not include any other checks – i.e., no configuration or signature checks. The historical analysis revealed that the above package names were indeed preloaded on the older devices and corresponded to the users of the API. However, in later versions, `"com.lge.onehandcontroller"` was removed, leaving the first path open to exploit. Observe that the other package, `"com.lge.systemserver,"` persisted in the later version but did not invoke the target API. We have confirmed that the Residual can be exploited to build a keylogger by simply squatting the removed package name. LG acknowledged and fixed this issue.

Keylogger on Samsung. We discovered through our historical analysis that Samsung has introduces an API `ISpenGestureService.getCurrentInputContext(...)` in 27 ROMs starting from version 7.0 through version 8.1. Our cross-model analysis revealed though that the API is used only by 8 ROMs; all from SM-N95x and SM-T82x series (corresponding to SNote and STab devices). Consequently, the API was flagged as a Residual in the rest of the 19 ROMs. We have manually investigated this case and found that the API can obtain an instance to an `IInputContext` object, maintained by the defining system service (i.e., `SpenGestureService`). `IInputContext` abstracts the input method to an app and allows reading, editing and controlling user inputs such as taps and hard key presses. Given these privileged operations, obtaining this object is restricted to the system and input method managers in other framework call sites. Our security analysis module revealed the Residual has no security checks at all, allowing any third-party app to get the `IInputContext` object with no permissions. We have confirmed that the Residual can be exploited to intercept all user input including lock screen passwords, payment data and app credentials. We have further confirmed that it can be exploited to inject and compromise the integrity of user inputs. Samsung confirmed and fixed the vulnerability. NIST ranked the vulnerability as critical.

Launching Activities with System Privilege. Our historical analysis discovered the presence of a Residual instance in the majority of our collected Samsung samples (versions 8.0 through 10). The API `ISemPersonaManager.startActivityThroughPersona(..)` was introduced and exclusively used in earlier Samsung devices running version 7.0. Our security analysis flagged the case as potentially vulnerable since it enforced obsolete access control. We inspected the Residual and surprisingly found that it allows starting any Android activity within the highly-privileged context of the defining system service (named `Persona`). Specifically, it takes as an argument any arbitrary intent describing the activity to be launched and invokes Android's `Context.startActivity()` to trigger the specified intent. This is clearly alarming since it can be exploited to trigger system activities without a privilege requirement.

We have built an end-to-end PoC for version 8.0 to demonstrate possible damages. For instance, we supplied an intent with action `"android.intent.action.ACTION_REQUEST_SHUTDOWN"` to trigger a system shutdown. In another instance, we crafted an intent to call emergency phone numbers (with an explicit destination to the package `"com.android.phone"` with data `"tel:911"`); all by exploiting the unnecessary Residual functionality.

Samsung marked this vulnerability as duplicate. The issue was previously reported and fixed.

9.2 Other Impacts

The impacts of Residuals are significant. Besides the end-to-end PoCs we built (Section 9.1), we randomly selected 250 reported weakly-protected Residuals and manually investigated potential consequences that could happen once they were exploited. We note that the instances here are randomly selected. We do not necessarily have a corresponding physical device, and the Residual implementation may correspond to undocumented proprietary functionalities. As such, all we could do is to statically inspect the code and *estimate* possible consequences once a Residual is invoked. Such an analysis may not be accurate, but it is still important for evaluating the impacts of weakly-protected Residuals that have never been investigated before. The results of our analysis are shown in Table 4. We group the possible impacts by category (first column) and give a few examples for each category (third column).

As shown in the table, 23 instances of Residuals can be exploited to expose (sensitive) user data. Particularly, we identified one instance that could be invoked to register a listener, allowing an

attacker to receive notifications of location updates. Other analyzed Residuals (18) allow manipulating data, including deleting cached files and other files under a specific directory. Our analysis further reveals 29 instances that can cause DoS attacks. One identified instance causes the device to deny and drop received SMS text messages. Another instance can be used to deny access to the external directory. We further identified other Residuals instances (34) that can be used to manipulate global settings, including Wlan configurations and SMS parameters. We could not predict the effect of 79 Residuals since they corresponded to undocumented proprietary features, while 67 other instances did not seem to lead to a clear security impact. As mentioned earlier, just like other APIs, weakly protected Residuals are not exploitable unless they implement a privileged functionality.

Table 4: Impacts of 250 Randomly-Selected Residuals

Impact	Count	Examples	Cause	Vendor(s)
Data leakage	23	Infer location	OAC	Samsung
		Get Mac address	OAC	Asus, Lenovo
		Read network variables	USF	Lenovo
		Infer running apps	USF	Huawei
Data pollution	18	Delete files under dir	USF	Xiaomi
		Delete cache files	OAC	Xiaomi
		Insert text message to ICC	OAC	Blu
DoS	29	Change subscription state	OAC	Xiaomi
		Deny SMS receipt	USF	Samsung
		Remount file system	OAC	Blu
Global setting manipulation	34	Change Wlan configuration	OAC	Xiaomi, Blu
		Change keyguard configuration	USF	Samsung
		Change audio output path	OAC	Xiaomi
		Change SMS parameters	USF	Blu, LG
Unclear – Undocumented features	79	Set Drx Mode	USF	Samsung
		Change cycle time	USF	Samsung
		Process AT Command	USF	Blu
		Infer ENDIP sample	OAC	LG
No Risk	67	–	–	–

OAC: Obsolete Access Control; USF: Undefined Security Feature

10 RELATED WORK

Android API Security Analysis. Developers frequently overestimate the permissions that their applications require as Android does not provide a complete permission specification. Earlier work has developed permission mapping strategies that can be incorporated into tools that detect application over-privilege and access control inconsistency. Stowaway [22] uses a feedback-directed testing approach to determine the maximum set of permissions that an Android application requires. PScout [14] relies on static analysis techniques to generate conservative permission mappings. Axplorer [16] improves upon previous permission mapping approaches by creating a static model of the Android framework that attempts to approximate the behavior of the threading mechanisms relied upon by framework services. The Arcade [12] tool goes a step further by generating a path-sensitive permission map that can be used to deduce an API's minimum required permissions. The current state-of-the-art permission mapping tool, Dynamo [20], uses a greybox fuzzing technique to generate path-sensitive permission mappings. We rely on these contributions to evaluate access control enforcement in Residual APIs using ReM .

Vendor Customization. Vendors tend to extensively customize device drivers, system applications and system services [27]. Since this customization is unregulated, it often introduces new security risks. Both Gallo et al. [23] and Possemato et al. [30] study the effect of vendor customization and find that it adversely affects Android device security. Zhou et al. [38] evaluate problematic vendor modifications to Linux device drivers. Harehunter [13] detects hanging attribute references, which can occur when customization results in references to nonexistent attributes that can then be defined by a malicious party. IPC Inspection [22] and ARF [25] focus on permission re-delegation vulnerabilities in Android applications and system services. Hay [26] analyzes the security of customized Android bootloaders. InVetter [37] identifies weakened input validation checks within customized system services. The Chizpurfle tool [19] uses a greybox fuzzing approach to uncover vulnerabilities in customized framework services. Zhang et al. [36] explore the impact of customization on the ION unified memory management interface used in ARM-based Android devices.

Evaluation of Customization/Custom APIs. Inconsistent access control at the API level, both within a single Android image and across images, can allow malicious third-party applications or users to perform privileged operations or access restricted resources [11]. Kratos [32] performs a path-insensitive static analysis to detect security policy inconsistency in application-accessible system services. AceDroid [11] models a wider array of security checks by transforming the checks into canonical security conditions and performing a path-sensitive analysis. ACMiner[24] takes another approach to inconsistency detection by relying on association rule mining. Our work is orthogonal in that it examines a specific category of Android APIs: Residual APIs, which are uniquely introduced by vendors but not used within applications or system services.

Dangers of Bloated Codebases. Most works exploring the security dangers/benefits of software debloating are limited to web applications. Azad et al. [15] explore the server-side, while Schwarz et al. [31] and Snyder et al. [33] focus on client-side browser security. Others, such as Mururu et al. [29] present binary debloating approaches, while Brown and Pande [17] [18] propose a new tool and metrics to assess the security impact of debloating. Ours is the first work to examine Residuals' impact on Android security.

11 CONCLUSION

As OEMs continue to customize AOSP codebases without regulation, the private OEM API lifecycle will inevitably lead to the persistence of Residual APIs. Our work is the first to center analysis of the OEM private APIs specifically on the security implications of Residuals. We analyze vendor Residual trends across time and models. We perform the first large-scale study of Residuals, covering 628 ROMs and spanning 7 vendors. We find that Residuals are pervasive and, more importantly, that they do indeed introduce serious access control vulnerabilities.

ACKNOWLEDGMENTS

This research was supported, in part by NSERC under grant RGPIN-07017 and by the Canada Foundation for Innovation under project 40236. This work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo. Any opinions, findings and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] 2020. The Heartbleed Bug. https://heartbleed.com.
[2] 2021. apktool. https://ibotpeaches.github.io/
[3] 2021. baksmali. https://github.com/JesusFreke/smali
[4] 2021. Dynamic Partitions. https://source.android.com/devices/tech/ota/dynamic_partition
[5] 2021. imjtool. http://newandroidbook.com/tools/imjtool.html
[6] 2021. lpunpack. https://github.com/LonelyFool/lpunpack_and_lpmake
[7] 2021. oat2Dex. https://github.com/testwhat/SmaliEx
[8] 2021. SALT. https://github.com/steadfasterX/SALT
[9] 2021. simg2img. https://github.com/anestisb/android-simg2img
[10] 2021. vdexExctractor. https://github.com/anestisb/vdexExtractor
[11] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. 2018. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection. Internet Society. https://doi.org/10.14722/ndss.2018.23121
[12] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. 2018. Precise android API protection mapping derivation and reasoning. In *Proceedings of the ACM Conference on Computer and Communications Security*. Association for Computing Machinery, 1151–1164. https://doi.org/10.1145/3243734.3243842
[13] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. 2015. Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1248–1259. https://doi.org/10.1145/2810103.2813648
[14] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 12. PScout: Analyzing the Android Permission Specification. In *CCS*. 1070.
[15] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1697–1714. https://www.usenix.org/conference/usenixsecurity19/presentation/azad
[16] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 48.
[17] Michael D. Brown and Santosh Pande. 2019. CARVE: Practical Security-Focused Software Debloating Using Simple Feature Set Mappings. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation* (London, United Kingdom) *(FEAST'19)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3338502.3359764
[18] Michael D. Brown and Santosh Pande. 2019. Is Less Really More? Towards Better Metrics for Measuring Security Improvements Realized Through Software Debloating. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/cset19/presentation/brown
[19] Domenico Cotroneo, Antonio Ken Iannillo, and Roberto Natella. 2019. Evolutionary Fuzzing of Android OS Vendor System Services. *CoRR* abs/1906.00621 (2019). arXiv:1906.00621 http://arxiv.org/abs/1906.00621
[20] Abdallah Dawoud and Sven Bugiel. 2021. Bringing Balance to the Force: Dynamic Analysis of the Android Application Framework, In Network and Distributed Systems Security (NDSS) Symposium 2021. *Bringing Balance to the Force: Dynamic Analysis of the Android Application Framework*. https://publications.cispa.saarland/3340/
[21] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2020. FIRMSCOPE: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2379–2396. https://www.usenix.org/conference/usenixsecurity20/presentation/elsabagh
[22] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. ACM, 726.
[23] Roberto Gallo, Patricia Hongo, Ricardo Dahab, Luiz C. Navarro, Henrique Kawakami, Kaio Galvão, Glauber Junqueira, and Luander Ribeiro. 2015. Security and System Architecture: Comparison of Android Customizations *(WiSec '15)*. Association for Computing Machinery, New York, NY, USA, Article 12, 6 pages. https://doi.org/10.1145/2766498.2766519
[24] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. 2019. ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware. (1 2019). http://arxiv.org/abs/1901.03603
[25] Sigmund Albert Gorski and William Enck. 2019. ARF: Identifying Re-Delegation Vulnerabilities in Android System Services. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks* (Miami, Florida) *(WiSec '19)*. Association for Computing Machinery, New York, NY, USA, 151–161. https://doi.org/10.1145/3317549.3319725
[26] Roee Hay. 2017. fastboot oem vuln: Android Bootloader Vulnerabilities in Vendor Customizations. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/woot17/workshop-program/presentation/hay
[27] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, and Cristina Nita-Rotaru. 2017. Chizpurfle: A Gray-Box Android Fuzzer for Vendor Service Customizations. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. 1–11. https://doi.org/10.1109/ISSRE.2017.16
[28] Li Li, Jun Gao, Tegawendé F. Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2018. Characterising Deprecated Android APIs. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 254–264. https://doi.org/10.1145/3196398.3196419
[29] Girish Mururu, Chris Porter, Prithayan Barua, and Santosh Pande. 2019. Binary Debloating for Security via Demand Driven Loading. arXiv:1902.06570 [cs.CR]
[30] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. 2021. Trust, but verify: A longitudinal analysis of Android OEM compliance and customization. In *S&P 2021, 42nd IEEE Symposium on Security and Privacy, 23-27 May 2021, Virtual Conference*, IEEE (Ed.).
[31] Michael Schwarz, Moritz Lipp, and Daniel Gruss. 2018. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_07A-3_Schwarz_paper.pdf
[32] Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. 2017. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. Internet Society. https://doi.org/10.14722/ndss.2016.23046
[33] Peter Snyder, Cynthia Taylor, and Chris Kanich. 2017. Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. arXiv:1708.08510 [cs.CR]
[34] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. 2013. The Impact of Vendor Customizations on Android Security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer amp; Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 623–634. https://doi.org/10.1145/2508859.2516728
[35] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, and Zhemin Yang. 2020. How Android Developers Handle Evolution-Induced API Compatibility Issues: A Large-Scale Study. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 886–898. https://doi.org/10.1145/3377811.3380357
[36] Hang Zhang, Dongdong She, and Zhiyun Qian. 2016. Android ION Hazard: The Curse of Customizable Memory Management System. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1663–1674. https://doi.org/10.1145/2976749.2978320
[37] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. 2018. Invetter: Locating insecure input validations in android services. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1165–1178.
[38] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. 2014. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *2014 IEEE Symposium on Security and Privacy*. 409–423. https://doi.org/10.1109/SP.2014.33

A FLAWS BREAKDOWN IN RESIDUAL AND ACTIVE APIS

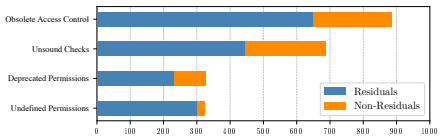

Figure 10: Flaws Breakdown in Residual and Active APIs