Scraping Sticky Leftovers: App User Information Left on Servers After Account Deletion

Preethi Santhanam, Hoang Dang
Wichita State University
{pxsanthanam1, hndang}@shockers.wichita.edu

Zhiyong Shan
Wichita State University
zhiyong.shan@wichita.edu

Iulian Neamtiu
New Jersey Institute of Technology
ineamtiu@njit.edu

Abstract

Sixty-five percent of mobile apps require user accounts for offering full-fledged functionality. Account information includes private data, e.g., address, phone number, credit card. Our concern is "leftover" account data kept on the server after account deletion, which can be a significant privacy violation. Specifically, we analyzed 1,435 popular apps from Google Play (and 771 associated websites), of which 678 have their own sign-up process, to answer questions such as: Can accounts be deleted at all? Following account deletion, will user data remain on the app's servers? If so, for how long? Do apps keep their promise to remove data?
Answering these questions, and more generally, understanding and tackling the leftover account problem, is challenging. A fundamental obstacle is that leftover data is manipulated and retained in a private space, on the app's backend servers; we devised a novel, reverse-engineering approach to infer leftover data from app–server communication. Another obstacle is the distributed nature of this data: program analysis as well as information retrieval are required on both the app and its website. We have developed an end-to-end solution (static analysis, dynamic analysis, natural language processing) to the leftover account problem. First, our toolchain checks whether an app, or its website, support account deletion; next, it checks whether the app/website have a data retention policy, and whether the account is left on servers after deletion, or after the specified retention period; finally, it automatically cleans up leftover accounts. We found that 64.45% of apps do not offer any means for users to delete accounts; 2.5% of apps still keep account data on app servers even after accounts are deleted by users. Only 5% of apps specify a retention period; some of these apps violate their own policy by still retaining data months after the period has ended. Experiments show that our approach is effective, with an F-measure > 88%, and efficient, with a typical analysis time of 279 seconds per app/website.

1. Introduction

A substantial percentage of mobile apps (65%, according to our findings) require user accounts. When creating a mobile app account, users have to provide private information, e.g., email address, phone number, billing address, or even SSN. Unfortunately, once this information has been gathered, only 35.55% of apps offer users the option to "forget" the information, e.g., offering a *Delete Account* option in app or on the website. Some companies deliberately make this process harder, e.g., users have to go to the company's website to delete a mobile-created account. Finally, some companies retain information even after users have asked for the account and associated data to be removed.

We denote *LAI* – **L**eftover **A**ccount **I**nformation – the account information retained on servers after account deletion. As server data is not (readily) accessible, our insight is to derive LAI from app-server interaction. We study how LAI is handled, from the perspective of user control: what information is required at account creation time; can users request account deletion; what information is retained after the user requests deletion, and for how long. LAI is problematic for two main reasons. First, LAI poses a security risk because leftover private information can be leaked after users have deleted their accounts. Second, LAI violates users' trust; users reasonably assume that account information is deleted when an account is deleted [37].

In Figure 1 we present several LAI examples. Figure 1(a) shows the PiniOn mobile app requiring name, password, birthday, email and gender at account creation time; Figure 1(b) shows the "account will be erased" warning shown when users request account deletion; Figure 1(c) shows that the app actually still keeps (at least some) account details on the backend servers after account deletion.

Less than a quarter of our examined apps (22.71%) provide users with means to delete their accounts, e.g., via an in-app 'Delete Account' button. Other apps (12.83%) do not provide this in-app option – instead they ask users to go to their corresponding websites and delete the accounts from there. Whether the company actually removes users' information after account deletion is a different story. For example, Figure 1 (d) shows how an account (anonymized) is still retained on eBay's servers after account deletion and *even after the 30 day-retention period claimed by the app*; in Section 9.2.1 we show other examples of popular apps that breach their promise to delete account data at the end of the retention period, keeping it for months after. In fact only 5% of apps specify a retention period (for how long data will be kept after account deletion): from 30 minutes to 5 years, typically 30 days. Finally, 437 apps (384 of which have more than 1M installs) do not provide any account deletion functionality, either in the app or on the website.

We performed a pilot study on 188 popular broad-ranging apps[1] from Google Play. Of these, 154 had a corresponding website; 135 ask for sensitive user information to create accounts. We manually installed each app, created an account, then explored the app and the website to check whether account deletion was supported. We deleted the account and uninstalled the app. We then re-installed the app and attempted to use the previous sign-up information upon account creation.

Table 1 shows the study results. We found that 34.81% of

1. Spanning 28 categories; involving free and paid apps; with app popularity ranging from 1M to 1B installs.

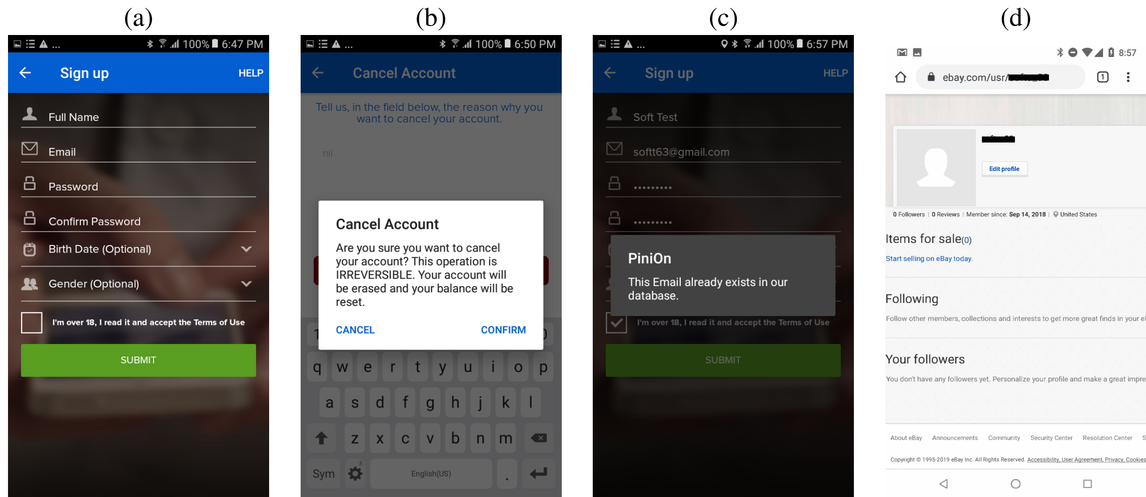


Fig. 1: Leftover Account Information: (a) Personal information required during PiniOn's sign-up; (b) App claiming the account will be "irreversibly" erased; (c) The key field of the account still retained on the server; (d) An user account (anonymized) still on eBay's servers even after the 30-day post-deletion retention period has expired.

the apps leave LAI on servers after app uninstallation, and 2.96% keep LAI after account deletion. A majority (57.04%) of apps do not offer a 'Delete Account' option, either in-app or on the website. Finally, for 23 apps, the account deletion process is convoluted (users must contact customer support, undergo phone-based verification, etc.). *Hence there is an impetus for an in-depth, systematic study of LAI.*

Challenges and approach. To our knowledge, there has been no effort so far to study and address LAI. A fundamental challenge is that data is stored privately, on the server, thus inaccessible to outsiders; we devised a novel approach based on reverse engineering to *infer* leftover data from app–server communication. Another challenge is the extensive scope of the analysis, even for one app: account management can be distributed across an app and its website, which requires both app and website analysis. Functionality described in natural language, in the app or website resources, e.g., XML or even images, has to be connected to account management actions in app code or website code. Deletion and retention policies might be deliberately "buried"; their precise retrieval and understanding is a challenging NLP task. We address these challenges via a four-tool chain – LeftoverAccountAnalyzer, AccountDeletionAnalyzer, LeftoverAccountCleaner (for uninstalled apps), RetentionPeriodAnalyzer – described in Sections 4 to 7. We made the tool implementations, datasets, and analysis results available on GitHub.[2]

We evaluate our approach, and perform an LAI study, in Section 9. We started from 1,435 Android apps and 771 corresponding websites; while 938 apps (65.4%) require account sign-up, 260 apps use third-party sign-in, hence the focus of the study was on the 678 "own sign-in" apps.

The study shows that our toolchain is effective. For example, LeftoverAccountAnalyzer found that 254 apps (37.46%) leave LAI after app uninstallation, and 17 apps (2.5%) leave LAI after account deletion, including government-issued IDs, or banking information (Table 7). Moreover, three apps with more than 50M installs keep LAI months after they were supposed to delete it (Table 9). We were able to confirm the LAI problem by contacting customer service, e.g., Enterprise Rent-A-Car. Finally, LeftoverAccountCleaner could successfully clean up accounts in 214 out of 245 uninstalled apps (87.34%). The study also shows that our toolchain is efficient: the median per-app analysis time for LeftoverAccountAnalyzer, AccountDeletionAnalyzer, RetentionPeriodAnalyzer and LeftoverAccountCleaner was 163 seconds, 276 seconds, 259 seconds and 231 seconds, respectively.

TABLE 1: Pilot Study Results (135 Apps).

App Info	Count	Percentage
LAI remains on servers		
after uninstallation	47	34.81%
after account deletion	4	2.96%
Account deletion function. (ADF)		
app & website	13	9.63%
in-app only, not on website	16	11.85%
on website only, not in-app	29	21.48%
no ADF	77	57.04%

Contributions. We make the following contributions:

- An exposition and study of the LAI problem.
- A novel, reverse engineering-based approach named LeftoverAccountAnalyzer to infer the account information remaining on servers after account deletion.
- An AccountDeletionAnalyzer tool to determine whether an app has account deletion functionality.
- A RetentionPeriodAnalyzer tool to automatically extract app retention period.

2. https://github.com/LeftoverAccountInformation/LAI

- A LeftoverAccountCleaner tool, to automatically clean up leftover accounts for a given Google user.
- An evaluation of the aforementioned tools on popular apps from Google Play.

2. Leftover Accounts: Problem Definition

The lifecycle of an account is shown in Figure 2. After an app is installed, the user signs-up (creates an account[3]) usually on the phone, or less frequently, on the app's website. The user's personal information is sent to the server and stored into the account database. When users decide not to use the app anymore, their options are to delete the account, uninstall the app, or both.

We believe that users should reasonably expect:

- Account deletion functionality, offered in the app or on the website.
- A retention policy that specifies for how long account data will be retained after account deletion.
- Account information to be removed from the server after account deletion; either immediately or after the policy-specified retention period.

Our pilot study shows that apps routinely violate these assumptions, leading to four categories of leftover accounts issues (in Sections 2.1 to 2.4 we define these categories from highest to lowest severity level).

LAI as security and privacy risk. LAI is a serious violation of user privacy [6] [1] [7]. Leftover accounts could get hacked, and the data in those accounts could be stolen or exposed. A server breach, e.g., years later, could expose information that users forgot they ever even shared.

2.1. No Account Deletion Functionality

Definition. An app does not provide users a means to delete their account; thus the user has no control over the private information that remains on the server side.

Example. Wattpad (wp.wattpad) is a popular app (100M+ installs) for reading or writing stories. However, neither the app nor the website provide users a 'Delete Account' option.

2.2. Leftover Accounts after Deletion

Definition. An app retains account information on the server side even after the user deletes the account.

Example. Discord (com.discord) is a popular (50M+ installs) communication app. First, we signed up for a Discord account via username/password. Next, we attempted to delete the account, by clicking the 'Delete Account' button. However, even months after this operation, we could confirm that LAI still remained on Discord servers.

2.3. Leftover Accounts after Uninstallation

Developers have the technical means to detect when their app is uninstalled: the uninstall event is available via the Google Firebase[4] app_remove event [24] or third-party libraries [12]. User expectations regarding account deletion upon app uninstall differ, depending on the type of the app. For "multi-platform services", such as Netflix or Hulu, delivered via either the browser or mobile app, users may assume that the account will persist upon app uninstallation. However, for "mobile-only services", i.e., services provided exclusively via the app, a vast majority of users assume that app uninstallation means the app is abandoned and thus the account is no longer needed [8] [21]. Note that mobile app retention rate is low: 75% of installed apps are abandoned within 90 days, and eventually uninstalled without being revisited [8] [21], which leaves users' information on servers (accounts becoming "zombie accounts"). If the servers are compromised, users' personal information will be exposed to hackers. Our toolchain determines whether an app actually deletes the user account upon app uninstall.

Definition. An app retains account data on the server after app uninstallation. If the app does not specify a data retention period, the data could potentially remain on the server for unlimited time. In our investigation, while certain apps have a post-account-deletion retention policy (how long data will be kept after account deletion), there were no apps with a post-app-uninstallation retention policy.

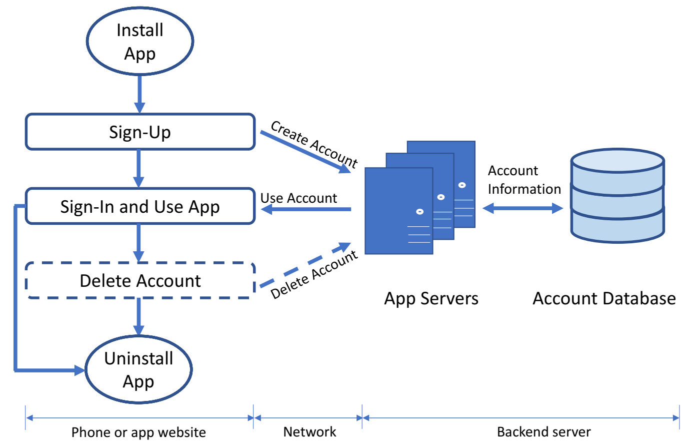


Fig. 2: Account Lifecycle.

2.4. No Account Retention Period

Definition. An app does not specify for how long the account (or account data) will be retained on the backend server after users request account deletion. For example Discord (com.discord)'s policy specifies that the account will be deleted "soon" but no firm period is provided [5].

3. Architecture

To detect and address LAI issues, we designed the toolchain shown in Figure 3. The toolchain performs a suite of static, dynamic, and NLP analyses on APK files, web pages, and

3. Some apps allow third-party sign-in (e.g., Google, Facebook); those apps are outside the scope of this paper.

4. Firebase is used by 99% of apps that employ a back-end [14] and 83% of apps overall – 2.5M out of 3M Google Play apps, as of October 2020 [13], [17], [22].

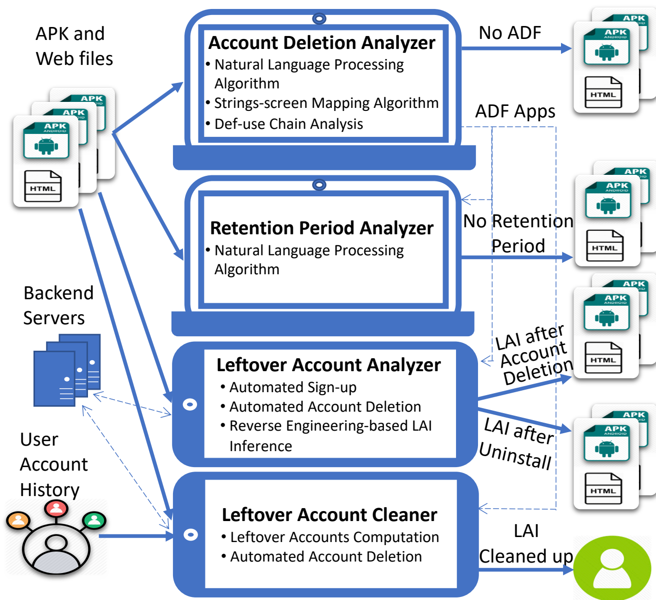


Fig. 3: Tool Chain Overview.

account history. The AccountDeletionAnalyzer statically analyzes APKs and web pages to determine whether the app (or its corresponding website) offers account deletion functionality. The LeftoverAccountAnalyzer uses dynamic analysis to detect and verify LAI after account deletion or app uninstall. The RetentionPeriodAnalyzer uses NLP on APKs and web pages to extract retention periods. The LeftoverAccountCleaner deletes accounts left after app uninstall.

4. Account Deletion Analyzer

This analyzer determines whether the app provides account deletion functionality (either in the app or on the website), e.g., via a button or link. Two challenges need to be addressed: (1) analyzing both the app binary (APK) and the app's corresponding website to find the GUI button or website link for account deletion; and (2) mapping the button/link to actual code, which requires an elaborate static analysis – note that GUI buttons are defined in the app's XML resources, whereas the actual code is defined in the app's Smali bytecode.

According to our observations: (1) the 'Delete Account', or similar, button/link has text whose semantics is account deletion and is bound with an action listener that performs the actual delete operation; and (2) the GUI layout hierarchy contains text to explain the consequences or steps of the account deletion process. We provide two examples of how websites encode **A**ccount **D**eletion (AD) buttons; a static HTML button on the left, and JavaScript code on the right:

HTML	JavaScript
<!–– www.curiosity.com ––> <div **class**="btn btn–danger"> <button **class**= "js–delete–account"> Delete Account </button> </div>	<!–– www.bodbot.com ––> <div **class**="input_holder"> delete my account </div>

To account for these aspects, AD analysis consists of four steps: finding account deletion strings, mapping strings to screens, finding action listeners, and determining account deletion functionality (ADF); each step is discussed next.

4.1. Finding Account Deletion Strings

Account deletion semantics – a precise approach for finding text whose semantics is "Delete Account" – is a major challenge due to three main reasons.

First, a context-free grammar is required to recognize AD strings. A less sophisticated approach, such as regular expressions, would not work. For example, suppose we use the regular expression "disable.* account" for AD string detection, and we analyze the string "sorry, online ordering services have been disabled for this account due to suspicious behavior". This string will match the regex, hence the phrase would *incorrectly* be deemed an AD string (it is not). Second, strings are scattered in hundreds of files across an app and its corresponding website, e.g., .xml resources, .smali (for code), or .html. Third, some strings are embedded in images, which requires converting the images to text before string analysis. We first perform keyword search to find candidate strings, which can dramatically reduce the number of strings to be handled by the natural language processing algorithm. Example patterns include ".*delete.*account.*", ".*close.*account.*", ".*cancel.*account.*", etc.

For strings that match the patterns, we developed a novel, natural language-based analysis approach to check whether the strings mean "delete account"; we name such strings *AD strings*. The grammar is defined as follows:

$$
\begin{aligned}
ADstring &::= Verbphrase\ Nounphrase \\
Verbphrase &::= \text{``}delete\text{''} \mid \text{``}destroy\text{''} \mid \\
&\quad\ \ \text{``}close\text{''} \mid \text{``}terminate\text{''} \mid \text{``}shutoff\text{''} \mid \\
&\quad\ \ \text{``}shutdown\text{''} \mid \text{``}disable\text{''} \mid \ldots \\
Nounphrase &::= Det\ Noun \mid Ppr\ Noun \mid Noun \\
Det &::= \text{``}the\text{''} \mid \text{``}this\text{''} \\
Noun &::= \text{``}account\text{''} \mid \text{``}registration\ information\text{''} \\
Ppr &::= \text{``}your\text{''} \mid \text{``}my\text{''}
\end{aligned}
$$

We use phrase structure trees, generated by NLTK [2], to reconstruct the semantic structure [10] of a sentence. For example (trees shown in the Appendix, Section 13.2), Fitbit's "please provide your password in order to delete this account" is in the language induced by the grammar hence an AD string (AD verb, $Nounphrase$ subtree whose $Noun$ is an AD noun), whereas Zomato's "sorry, online ordering services have been disabled for this account due to suspicious behavior" does not conform to the grammar hence is not an AD string (succeeding subtree of the $Verbphrase$ is a propositional phrase tree as opposed to $Nounphrase$).

Tables 2 and 3 show AD strings vs. non-AD strings discerned by our grammar, which illustrates the difficulties and subtlety of the task, along with our approach's effectiveness.

4.2. Mapping Strings to Screens

There is a disconnect between strings' *definition* (location) and *use* (where exactly, in the GUI, the strings are actually

TABLE 2: AD String Examples.

Package Name	Phrases
adidas	Delete account
com.foap.android	We are sorry you want to close your account.
com.airbnb.android	Cancel your account?
com.azarlive.android	Are you sure you want to delete your account?
com.bearpty.talklife	Do you want to delete your account?
com.clue.android	How do I delete my account?

TABLE 3: Non-AD String Examples.

Package Name	Phrases
microsoftword	To see the Microsoft Certification data that's linked to this account, and your friends list will all be deleted.
com.bt.mdd	You are required to be logged in to delete a book from your account
com.feverup.fever	Clicking below will delete all the payment methods linked to your Fever account
com.goldstar	By tapping confirm, your tickets will be canceled, and your Goldstar account will be credited the amount above
com.huawei.health	Cancel logging in with this account?
com.penzu.android	This entry will be deleted from your device and anywhere else you use to access your Penzu account

shown to the user). For example, AD strings can be embedded in app resources (XML files or images), which cannot be directly connected to a screen; here by "screen" we mean app activities (pages), or web pages on the app website. We address this disconnect challenge via a novel, static analysis-based approach. We use static analysis as it has several advantages over dynamic analysis for this setting: since we do not have to run the app, the analysis is efficient, scalable, and sound. However, static mapping is challenging, because we cannot *directly* map an AD string to a screen, as there are multiple intermediate mapping steps. Typically, intermediate steps include mapping strings to name attributes, then UI controls, then layouts, then top-level layouts, then fragments, and finally, activities. Different intermediate steps involve different objects, which requires tracking via static analysis.

All potential intermediate mapping steps and objects of an app form a directed object graph: nodes represents objects (i.e., strings, name attributes, UI controls, images, layouts, top-level layouts, fragments, activities, and HTML files), while edges represent intermediate mapping steps. We thus *reduce the problem of mapping an AD string to a screen* to *finding a path from the string to the screen in the graph*. As there are thousands of objects in a decompiled app and millions of potential intermediate mapping steps among these objects, finding a path from a string to a screen in this graph is inefficient. To reduce intermediate mapping steps and improve

Algorithm 1 Mapping AD String to Screen

Input: ADString

```
 1: procedure MappingStringToScreen(adString)
 2:     objType ← Str
 3:     screenSet ← DepthFirstMap(adString, objType)
 4:     return screenSet
 5: end procedure
 6:
 7: procedure DepthFirstMap(obj_i, type_i)
 8:     objTypeSet ← GetMappingObjectTypes(type_i)
 9:     for each type_j in objTypeSet do
10:         objSet ← FindMappingObjects(obj_i, type_j)
11:         for each obj_j in objSet do
12:             if obj_j is activity or HTML file then
13:                 screenSet ← screenSet + obj_j
14:             else
15:                 screenSet ← DepthFirstMap(obj_j, type_j)
16:             end if
17:         end for
18:     end for
19:     return screenSet
20: end procedure
```

performance, we instead enumerate legal/feasible intermediate mapping steps:

$$Strings \dashv (UIControls \mid NameAttributes \mid Images \mid HtmlFiles)$$
$$NameAttributes \dashv (UIControls \mid Layouts \mid Fragments \mid Activities)$$
$$UIControls \dashv Layouts$$
$$Layouts \dashv TopLevelLayouts$$
$$TopLevelLayouts \dashv (Fragments \mid Activities)$$
$$Fragments \dashv Activities$$
$$Images \dashv (UIControls \mid HtmlFiles)$$
$$Activities \dashv \phi$$
$$HtmlFiles \dashv \phi$$

There are 15 types of possible intermediate mapping steps (i.e., containing relationships), denoted as '$\dashv$'. For example, "Strings $\dashv$ UIControls" indicates that strings can be contained by UIControls. Activities and HTML files are screens and thus not contained by any objects. With these 15 possible intermediate mapping steps, we design a depth-first algorithm to map an AD string to screens (Algorithm 1). The function GetMappingObjectTypes uses the mapping table above to determine the object types that can contain the given object type. This can avoid searching a large volume of objects with other object types, thus improving performance. The function FindMappingObjects gathers all objects that contain the given object. Once the mapping procedure reaches a screen, i.e., activity or HTML file, we have determined that the screen contains the string.

Figure 4 uses the Line app as a complete example to illustrate the intermediate mapping steps from AD strings to screens. The app has 3 AD strings (Adstr) on the top of the graph. These AD strings are mapped to their respective string name attributes (Nattr). The attributes are used in two layouts (Lay), contained in one top-level layout (Toplay). The

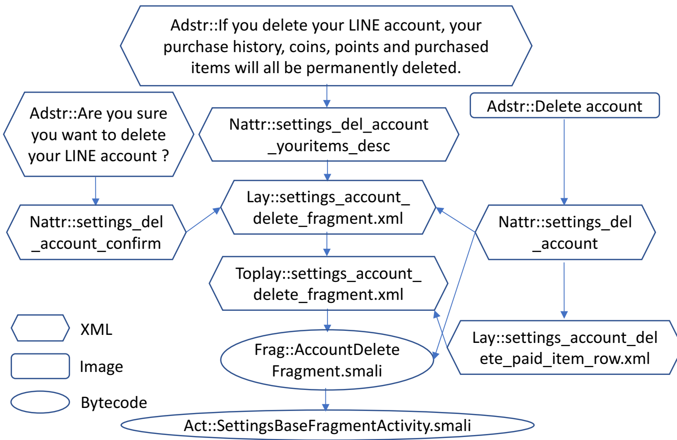


Fig. 4: String-screen Mapping Graph for Line App.

top-level layout is used by a fragment (Frag). Finally, the fragment is used by an activity (Act). Hence we successfully mapped the AD string "If you delete your LINE account..." to Android activity SettingsBaseFragmentActivity.

4.3. Finding Account Deletion Listeners

We reconstruct the link from AD code to AD GUI elements via def-use-chain analysis: specifically, we connect AD strings shown on a screen with GUI actions the user can perform on that screen. Def-use-chain analysis has so far been applied to program code, to connect variable definitions with variable uses. In Android apps, however, many definitions are in configuration files, e.g., strings, or GUI element IDs, which are outside the purview of traditional def-use-chain analysis. We now discuss our approach for performing this analysis across program code and configuration files.

In Android, GUI actions such as button clicks are passed to app code via listeners, e.g., onClickListener(). To associate a button with the action, the app needs to bind the listener with the button when the activity is created. This is done by calling set*Listener-style functions, e.g., setOnClickListener, setOnKeyListener, setOnTouchListener. If the action listener can be traced, via def-use-chain analysis, to one of the AD component id numbers, we conclude that Account Deletion Functionality (ADF) has been found. The following code snippet is from EyeEm, a popular photo app.

```
1 const v1, 0x7f090322
2 invoke−static {p2, v1, v0}, Lbutterknife / internal / Utils ;→
      findRequiredView(Landroid/view/View;ILjava/lang/String;)
3 move−result−object v0
4  ...
5 invoke−virtual {v0, v1}, Landroid/view/View;→
      setOnClickListener(Landroid/view/View$OnClickListener;)
```

On line 5, setOnClickListener is invoked on v0. We use def-use chains to walk backwards to its definition: v0 is defined on line 3, and assigned the value returned by findRequiredView. We continue to use def-use chains to find the definition of findRequiredView's second parameter, i.e., 0x7f090322. The parameter value is defined as the id of settings_delete_account_confirm in the layout (public.xml).

```
1 <public type="id" name="settings_delete_account_confirm" id="0
      x7f090322" />
```

Hence we conclude that AD button settings_delete_account_confirm is linked to AD code onClickListener. Note that, as the button id definition resides in the public.xml configuration file, extending the def-use chain analysis beyond bytecode was a key enabler for our approach.

4.4. Determining ADF

ADF is determined by two conditions: (1) a layout associated with an action listener shows text with account deletion semantics; and (2) on the same screen, there are other GUI components containing text with account deletion semantics. These components help users understand the action, or consequences respectively, of deleting the account.

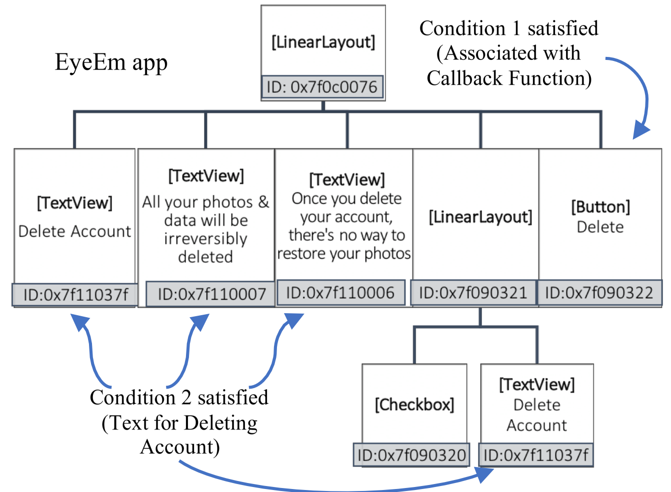


Fig. 5: An Example for Determining ADF.

Figure 5 shows an example of an account-deletion top-level layout from the EyeEm app. In the figure, fragment frag_settings_delete_account.xml is represented as a hierarchy, with a [LinearLayout] root node, ID 0x7f0c0076. There are three descendant [TextView] nodes; the interesting one is ID 0x7f11037f, showing the text 'Delete Account'. This satisfies condition (2). Also, in the figure, ID 0x7f090322 is mapped to node [Button] with text 'Delete'. We showed that this Button has a listener from Section 4.3, which satisfies condition (1). Hence the layout is an ADF layout.

5. Retention Period Analyzer

Some apps have a well-defined *retention policy*: account information will be retained on the server for a specified period of time (e.g., 30 days) after users request account deletion. We constructed an analyzer that checks for, and extracts, this retention period. In our evaluation (Section 9.1), to confirm whether the account information was indeed removed after the retention period had passed, we waited for the end of retention period, then checked for LAI.

To automatically extract the retention period, RetentionPeriodAnalyzer finds text that meets two conditions. First, the

TABLE 4: Retention Period String Examples.

Package Name	Phrases
com.snapchat.android	After **30 days**, your account will be deleted
com.opera.browser	Your account will be deleted in **7 days**
com.azure.authentic.	This account becomes unrecoverable **60 days** after you close the account
com.intsig.camscan.	The account will be deleted in **14 days** after we receive your email
com.ebay.mobile	We have started the process to close your account. The process may take up to **30 days**
net.wargaming.wot.b.	**45 days** after creating a request, your account will be deleted forever

text must contains strings with Account Deletion or Account Restoring semantics (similar to AD semantics in Section 4). Second, the text must contains a time period string, consisting of a number and a time unit; for example, "1 month", "3 days", or "30 minutes". This time period is the retention period. The grammar is defined as follows:

$$
\begin{array}{lll}
RetentionPeriodString & ::= & In\ Nounphrase \mid \\
 & & Nounphrase\ In \mid \\
 & & To\ NounPhrase \\
In & ::= & \text{“}after\text{”} \mid \text{“}in\text{”} \mid \text{“}within\text{”} \\
Nounphrase & ::= & Cd\ Nns \mid Cd\ Jj\ Nns \\
Cd & ::= & Integer \\
Nns & ::= & \text{“}minutes\text{”} \mid \text{“}hours\text{”} \mid \text{“}days\text{”} \\
 & & \mid \text{“}weeks\text{”} \mid \text{“}months\text{”} \\
To & ::= & \text{“}to\text{”} \\
Jj & ::= & Adjective
\end{array}
$$

Table 4 shows six examples of retention period strings identified by the grammar, which illustrates that our grammar is effective for recognizing retention periods.

6. Leftover Account Analyzer

To find whether an app leaves leftover account data we use an automated approach enabled by Appium. For LAI after account deletion, LeftoverAccountAnalyzer performs several steps, as described shortly: sign-up (account creation), account deletion (where offered), sign-up/create account again with the same credentials, and LAI verification. LAI after app uninstallation follows the same steps, except the account deletion step is replaced with app uninstall and app reinstall. We now describe each step.

Initial sign-up. LeftoverAccountAnalyzer automatically installs the app, finds the 'Sign-up' screen, creates a new account by supplying the required user information, potentially navigating through multiple screens via 'Next' or similar, until successful sign-up is confirmed. In the process, LeftoverAccountAnalyzer logs the {GUI element→text input} mappings, which are key to LAI detection.

Note that automating this process was far from trivial, due to the aforementioned issues (text embedded in images, sign-ups over multiple pages) and additional technical challenges, as described next. To ensure correct processing, we used a hybrid process (automated, but with human oversight), as follows. We manually collected the word sets used in the 135-app pilot study to mark $Sign\text{-}up \rightarrow Next \rightarrow [Done \mid Error]$ stage transitions. Then, we used those sets to seed the automated approach and monitored LeftoverAccountAnalyzer's workflow on each app. Apps that deviated from the expected workflow (e.g., used anti-automation techniques, as explained shortly) were routed for manual analysis; there were 39 such apps.

Specifically, we use three word sets as "anchors": sign-up, next, and done. The *sign-up* word set contains keywords such that users can click on their respective widgets to start the account sign-up (e.g., "sign up", "register", "create account", "get started", "join", "continue with email", "register with email", "log in", "register", etc.). The *next* word set contains keywords that help identify next-screen or next-page transitions; users can click on the associated widgets to move to the next screen (e.g., "next", "ok", "continue", "advance", "move forward", "click", "skip", etc.). Finally, the *done* word set is associated with widgets used to finish the account sign-up (e.g., "finish", "done", "complete", "create account", "set up", "verify phone number", "verify email address", "sign up", "sign in", "log in", etc.).

Manual intervention was needed in three scenarios: when apps used CAPTCHAs; to verify the phone number or email address; and for confirmation prompts (privacy policy, upgrading membership, enabling location, etc).

Automated account deletion. LeftoverAccountAnalyzer invokes AccountDeletionAnalyzer to determine whether an app has an AD button; if so, LeftoverAccountAnalyzer will note the name of the corresponding screen and perform the following workflow to automatically delete the leftover accounts, created by the app. The workflow consists of three stages, *Login → Finding ADF → Deleting the Account*, which are navigated and exercised automatically.

Login. The log-in screens are anchored by word sets shown on clickable widgets such as "log in", "signin", "log in with email", "sign in via email", "login or register", "existing user", "I already have an account", "get started", etc. The LeftoverAccountAnalyzer fills in the login information using the sign-up information retained from Step 1.

Finding ADF. The analyzer navigates to the screen containing ADF widgets (e.g., Delete Account buttons) by using a word set which includes "more options", "profile", "account settings", "edit profile", "my account", etc. If the current screen matches the screen name found by the AccountDeletionAnalyzer, it moves to the next stage.

Deleting the Account. The analyzer completes the deletion process by finding and clicking the deletion widget ("delete account", "close my account", etc.).

Repeating the sign-up. LeftoverAccountAnalyzer attempts to repeat the sign-up: installs the app, extracts the GUI layout, and injects the same, Step 1, input values into the GUI. Apps are categorized into apps that allow a second sign-up, and apps that display error messages, e.g., "account cannot be created

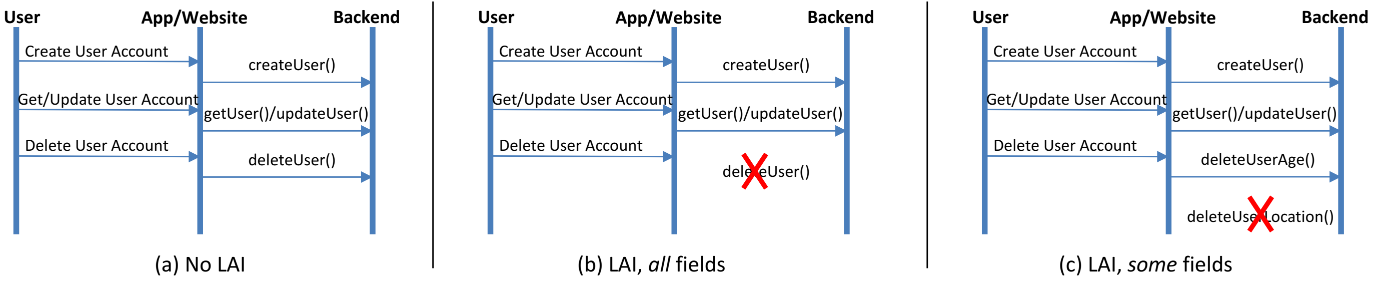


Fig. 6: Message sequence chart for No LAI (left) vs. LAI-all fields (center) vs. LAI-some fields (right).

because it already exists in the system." *We can thus check whether the account is retained on the server, automatically and at scale.* Next, we show how we identify the precise extent, i.e., fields, left on the server.

Verifying LAI. All fields entered when signing-up are potentially retained on the server (hence constitute LAI). Confirming this retention by examining sever-side information is infeasible, as we do not have access to apps' (backend) servers. Hence we developed an approach to infer retained fields, as explained next.

At high level, our approach detects the three scenarios shown in Figure 6. In each scenario, we show the interaction between user, app/website, and backend, for account creation, account update, and account deletion. The actual deletion happens in the communication between the app/website and the backend. On the left, we have the message sequence that ensures account deletion: a request to delete the account is reflected in a deleteUser() being sent to the backend. In the middle, there is no such API call, hence *all fields* are LAI. On the right, we have an API call to delete the user's age, i.e., deleteUserAge() but no API call to deleteUserLocation(), meaning the user's location will be left on the server. Next, we show our approach for inferring such LAI.

6.1. Reverse Engineering-based LAI Inference

We reverse-engineer the app–backend server communication to infer LAI by tracing and analyzing the AM (Account Manipulation) operations; these operations are initiated in the app but ultimately create, delete, or change user accounts in the backend database.

There are four types of AM operations (two coarse-grained and two fine-grained): account creation and deletion that manipulate *the whole account*, as well as field creation and deletion that manipulate *a specific field* of an account. These AM operations are implemented via API functions provided by backend software development kits (SDK) or via requests to a database.

To trace AM operations, we decompile apps' APK files into Smali code; identify account creation/deletion/manipulation methods; extract AM API calls in these methods and their call graphs' transitive closure; and intercept AM calls to log their in/out parameters at runtime. An alternative approach to reverse-engineering AM calls would be to intercept traffic, e.g., via mitmproxy [16]; however, messages would still ultimately be "put on the wire" via the AM API calls, hence our approach (AM API call interception) would reveal those messages.

Table 5 lists the AM API functions we logged and the corresponding AM operations. To comprehensively cover AM operations, we analyze both *remote* and *local* AM operations. We focused on the Firebase (including CloudFirestore and FirebaseDatabase) and Parse backends due to their popularity, e.g., 99% of apps that employ backend SDKs use Firebase [14].

Local operations involve an additional step: rather than communicating with Firebase/Parse directly, local AM operations write into Android's SharedPreferences or Bundle; data is sent from local stores to the backend when necessary. Table 5 omits these operations for brevity.

User account data can be stored into two separate locations on the backend server. The *primary location* is controlled by (hardcoded into) the backend SDK, but it only supports five fields per account, which makes it unsuitable for most applications. A *custom location* stores extra information of an account at a location not controlled by the backend SDK. For example, a custom location can be a separate node under /users, inside the server database itself.

We run the app (with AM calls intercepted) to create and delete an account. From AM logs we determine the fields retained on the backend servers as follows:

$$F_r = (F_{ap} + F_{ac}) - (F_{dp} + F_{dc})$$

where F_r, F_{ap}, F_{ac}, F_{dp} and F_{dc} represent fields retained, added in the primary location, added in the custom location, deleted in the primary location and deleted in the custom location, respectively.

As a first example, the app Rent-A-Car stores the user's first name, last name, email, password and phone number in its primary location; and street address in its custom location. Therefore F_{ap} = {first name, last name, email, password, phone number}, F_{ac} = {street address}. The app does not delete any fields when deleting an account, hence $F_{dp} = F_{dc} = \emptyset$ and F_r = {first name, last name, email, password, phone number, street address}.

As a second example, consider the PiniOn app (which helps brands and businesses understand their market and consumers). The app's code for creating a new user account is shown in the following code snippet; hence F_{ap} = {name, email, password}.

TABLE 5: APIs for AM Operations.

AM Operations	Account Creation	Account Deletion	Field Creation/Deletion
Firebase	createUser, createUserAsync, createUserWithEmailAndPassword	delete, deleteUser	updateUser, updateEmail, updatePassword, updatePhoneNumber, updateProfile
CloudFirestore	add	delete	set, update, delete
FirebaseDatabase	setValue	removeValue	updateChildren, setValue
Parse	signUp, signUpInBackground	delete, deleteInBackground, deleteEventually	setUsername, setEmail, setPassword, put

```
1   .param p1, "name"    # Ljava/lang/String;
2   .param p2, "email"    # Ljava/lang/String;
3   .param p3, "password"    # Ljava/lang/String;
4   .param p4, "handler"  # LFirebase$ResultHandler;
5   ......
6   invoke-virtual {v0, p1, p2, p3, p4}, Lcom/firebase/client/
        authentication/AuthenticationManager;→ createUser(
        Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;
        Lcom/firebase/client/Firebase$ResultHandler;)V
```

The app adds two extra fields in the following two code snippets. Therefore, F_{ac} = {birthday, gender}.

```
1   const-string v3, "birthday"
2   ......
3   invoke-interface {v1, v3, v4}, Landroid/content/
        SharedPreferences$Editor;→ putString(Ljava/lang/
        String;Ljava/lang/String;)Landroid/content/
        SharedPreferences$Editor;
```

```
1   const-string v3, "gender"
2   ......
3   invoke-interface {v1, v3, v4}, Landroid/content/
        SharedPreferences$Editor;→ putString(Ljava/lang/
        String;Ljava/lang/String;)Landroid/content/
        SharedPreferences$Editor;
```

As there is no AM delete operation, $F_{dp} = F_{dc} = \emptyset$. Finally, the LAI is: $F_r = (F_{ap} + F_{ac}) - (F_{dp} + F_{dc})$ = {name, email, password, birthday, gender}.

7. Leftover Account Cleaner

Account cleanup is a substantial user burden if it has to be performed on dozens of uninstalled apps. To study the feasibility of "automatic cleaning" we built LeftoverAccountCleaner, a tool that helps Android users automatically delete leftover accounts after app uninstall. LeftoverAccountCleaner runs on a laptop, connected to the phone via `adb`.

The approach has three steps. First, LeftoverAccountCleaner finds uninstalled apps as the set difference between the apps in the account history on Google Play Store and the apps currently installed on the phone (the account history records all apps downloaded by the user, including the apps that have been uninstalled). Second, LeftoverAccountCleaner re-installs the uninstalled app by downloading it from Google Play.[5] Third, LeftoverAccountCleaner deletes the leftover account incurred by the formerly-installed apps by logging-in, finding ADF, and finally deleting the account, leveraging some LeftoverAccountAnalyzer modules (Section 6).

5. In our experiments, the same version of an uninstalled app was available on Google Play. If an uninstalled app is an older version compared to the current version on Google Play, retrieving and installing the older version does not pose a substantial challenge, as in that case the older app version is available on app mirroring sites, e.g., https://www.apkmirror.com/.

8. Implementation

Our toolchain is implemented in Python, Java, and shell script – about 6,000 LOC in total. The toolchain relies on several "enabler" tools and libraries. To decode and rebuild APK files, we employ apktool [3], a tool for reverse engineering third-party, closed-source Android apps. As important strings can be stored in images, we use Tesseract OCR [11] to convert all images embedded in app screens and web pages into strings. We use Screaming Frog [9] to crawl both static and dynamic web content. We leverage the Appium mobile automator [4] for UI and user interaction automation. We employ NLTK [2] to support natural language processing. To enable static data flow analysis (i.e., def-use chain analysis), we have modified Flowdroid [19]. To intercept and log AM calls we built an LAI Monitor (described in Section 13.1) that leverages the Xposed Framework [15] to "hook" Android API functions.

9. Evaluation

We now present the results of a study of LAI issues, as well as an evaluation of our toolchain. We start by describing the dataset and test environment. Next, we present the high-level findings (Section 9.1). Finally, we evaluate our approach along two dimensions – *effectiveness* (Section 9.2) and *efficiency* (Section 9.3).

Dataset. We started with 1,435 APKs from Google Play Store and 771 corresponding websites; some apps do not have websites. We only selected Android apps with a high number of installs (more than 500K). The set was chosen to cover a wide range of categories (Sports, Business, Communication, Health and Fitness, Education, Games, etc.) and includes both free and paid apps. Among the 1,435 apps, 938 apps require user information when creating accounts (the other 497 do not require sign-in); of those 938 apps, 260 use third-party sign-in. Therefore, we focused on the remaining 678 apps which use their own accounts and collect user information. The 188 apps used in the pilot study are not included in the 1,435-app dataset.

Environment. Our testing environment consisted of: an Intel Xeon server (36 logical cores@4.3 GHz, 128GB RAM; Ubuntu 18.04) where we ran the ADF analysis; 2 MacBook

TABLE 6: Statistics on the 678-apps Dataset.

App Info	Count	Percentage
LAI remains on servers		
after uninstallation	254	37.46%
after account deletion	17	2.5%
ADF buttons		
app & website	38	5.6%
in-app only, not on website	116	17.11%
on website only, not in-app	87	12.83%
no ADF buttons	437	64.45%

TABLE 7: Account information retained (LAI, middle columns) or collected (last column).

LAI Category	#Apps with LAI After Uninstallation	#Apps with LAI After Account Deletion	#Non-ADF Apps
Government ID	6	0	12
Banking Info.	49	4	94
Password	218	15	418
Last Name	101	7	202
First Name	107	7	210
User Name	66	6	135
Phone	181	11	352
Email	230	16	437
ZIP Code	27	0	48
Location (address or GPS)	210	15	423
Photo	89	3	189
Gender	80	6	139
Weight	25	3	51
Height	41	3	66
Age	82	9	148
Social Network	180	10	357
Language	73	7	140

Pro laptops (core i5, 8GB RAM) used to configure the Appium server and to run LAI analysis as well as the LeftoverAccountCleaner; and 2 Android phones (4 ARM cores@1.40 GHz, 2GB RAM, 16GB storage, Android 6.0).

9.1. High-level Findings

The high-level findings are shown in Table 6: 254 apps (37.46%) have LAI after app uninstallation; 17 apps (2.5%) have LAI *after account deletion*; and 437 apps (64.45%) *offer no ADF functionality at all*. We believe that keeping LAI after deletion, as well as not offering users a 'Delete Account' option are substantial reasons for concern. These findings corroborate our pilot study's findings (Table 1).

9.2. Effectiveness

This section discusses the effectiveness of, and analysis results for, each of the four tools in our toolchain.

9.2.1. Leftover Account Analyzer. This analyzer infers the information left on the servers, i.e., LAI. Table 7 shows the number of apps that our analysis flags as leaving sensitive information after app uninstallation; after account deletion; and information collected by apps that do not offer ADF (discussed in Section 9.2.2). Certain LAI categories consist of related fields grouped together, e.g., 'Government ID' (SSN, driver license number, employee IDs) and 'Banking Information' (credit card number, CVV number, payment methods, transaction details, billing information).

LAI after uninstallation. As shown in the second column of Table 7, a large number of apps have LAI after app uninstallation. While some apps keep LAI in the name of convenience, e.g., users might want to "re-activate" their account, a clear retention policy should still be in place to indicate for how long data is retained. We found that email and password are the most common types of leftover information (230 apps require email, 218 apps require a password), followed by location (210), social network profiles (181), and phone number (180). Government IDs and banking information are the least common, but the most sensitive LAI: 6 apps require driver license (or social security numbers) and 49 apps require banking information (credit card number, payment method, billing address).

Some notable apps which have LAI after app uninstallation include: Microsoft Word (email address), Spotify (social network accounts), WhatsApp (phone numbers), AdobePhotoshop (edited photos), eHarmony (spoken languages), McDonald's (GPS location), MyFitnessPal (weight and ZIP code).

LAI after account deletion. The third column of Table 7 shows that email, password, (and concerningly) location and social network profiles, are the most frequent LAI left after account deletion – more than 10 apps kept this information post-account deletion. We verified whether apps retained data in two scenarios. First, for apps *without* a retention period (Table 8) we deleted the accounts on 02.01.20, then checked for LAI on 02.26.20 and again on 07.10.20. Second, for apps *with* a retention period (Table 9) we deleted some accounts on 02.01.20, some on 02.20.20, then checked for LAI *after the longest possible retention period had ended*: on 04.16.20 and again on 07.10.20.

As shown in Tables 8 and 9, 17 apps (2.5%) have LAI even after account deletion. Of the 17 apps, 15 have more than 1M installs. Apps in Table 8 do not specify a retention period, but never truly remove account information even after contacting their customer support, e.g., the OLIO app. Apps in Table 9 have a specified retention period (7 minutes–30 days). However, *when we rechecked the apps after the periods expired, the accounts were still not deleted*, which violates users' trust and the app's own policy.

We contacted the companies whose apps appear in Table 8 and Table 9 by a variety of means: 'Live chat', 'Contact' form, and email. Most companies have agreed to remove the leftover account data. One company (Fitbit) informed us that they would like to keep a part of the user data. Finally, some companies have asked us to first submit official identification documents (passport, driving license) before proceeding to remove data.

9.2.2. Account Deletion Analyzer. Ground Truth was obtained via manual analysis on the 678 apps, which involved

TABLE 8: Apps WITHOUT a Specified Retention Period.

Package Name	#Installs	Account Deletion	Checked On	Rechecked On	Data Retained (or other LAI Confirmation)
com.bandainamcoent.google.pac	5M	02/01/2020	02/26/2020	07/10/2020	email, password
com.olioex.android	1M	02/01/2020	02/26/2020	07/10/2020	customer support has not deleted the account
com.discord	50M	02/01/2020	02/26/2020	07/10/2020	still able to restore account
com.bearpty.talklife	500K	02/01/2020	02/26/2020	07/10/2020	email, password
com.myfitnesspal.android	50M	02/01/2020	02/26/2020	07/10/2020	username
com.relayrides.android.relayrides	1M	02/01/2020	02/26/2020	07/10/2020	email, first name, last name, password
com.ehi.enterprise.android	1M	02/01/2020	02/26/2020	07/10/2020	full name, email, password, address, phone
com.tophatter	10M	02/01/2020	02/26/2020	07/10/2020	email, password, user name
com.etsy.android.soe	1M	02/01/2020	02/26/2020	07/10/2020	email address, first name, password
com.magix.android.mmjam	10M	02/01/2020	02/26/2020	07/10/2020	artist name, email, password
com.mercariapp.mercari	10M	02/01/2020	02/26/2020	07/10/2020	email, password, username
br.com.pinion	1M	02/01/2020	02/26/2020	07/10/2020	full name, email, passwd., DOB, gender
com.sarahah.com	1M	02/01/2020	02/26/2020	07/10/2020	full name, email, password, avatar

TABLE 9: Apps WITH a Specified Retention Period.

Package Name	#Installs	Account Deletion	Retention Period	Checked On	Rechecked On	Data Retained (or other LAI Confirmation)
com.pinterest	100M	02/01/2020	14 days	04/16/2020	07/10/2020	full name, age, gender, country, email, passwd
com.quora.android	10M	02/01/2020	14 days	04/16/2020	07/10/2020	first name, last name, email, password
com.fitbit.fitbitmobile	50M	01/20/2020	7 days	04/16/2020	07/10/2020	email, password, first name, last name, birthday, height, weight, sex
net.wargaming.wot.blitz	50M	01/20/2020	45 days	04/16/2020	07/10/2020	account can still be restored

TABLE 10: ADF Analysis Results.

	True	FP	FN	Precision	Recall	F-measure
Apps (678)	209	32	23	86.7%	90.1%	88.4%

2 raters and 100% agreement. We checked both the app and its corresponding website. An app was labeled as ADF if it offered an account deletion button or link, either in app or on the website. The first two authors, PhD students, performed several tasks independently: unpacked the apps and analyzed their binaries, ran the tools, checked the app-server communication, etc. Next, they met to cross-check the findings – the findings were in agreement for all apps.

We confirmed 209 apps with ADF. AccountDeletionAnalyzer reported 241 ADF apps, over-reporting 32 apps (false positives) and under-reporting 23 apps (false negatives); the reasons for false positives and false negatives will be discussed shortly. Table 10 shows that the precision is 86.7%, while the recall is 90.1%; hence the F-measure is 88.4%.

These results allow us to conclude that AccountDeletionAnalyzer is effective.

For false positives, we identified two major reasons. First, an app could provide deletion functionality for accounts other than the user account. An example is Business Calendar (com.appgenix.bizcal, 5M+ installs). The app supports deleting birthday accounts, which does not delete the user account. Second, the app might set the deletion button as invisible. For example, Bleacher Report Live (com.bleacherreport.android.teamstream, 1M+ installs) contains an AD string, button, and listener, but its setupUI() method calls this.mDeleteButton.setVisibility(8) where setVisibility(8) renders the AD button invisible.

For false negatives, we identified three reasons. First, pre-NLP pattern matching might fail to find potential AD strings. For example, our analysis could not find the AD strings in the Goodreads book reviewing app (com.goodreads, 10M+ installs). Second, image conversion might fail to convert image AD text into plain text. For example, Guides by Lonely Planet (com.lonelyplanet.guides, 1M+ installs) has ADF but our tool failed to convert its AD string images into text. Third, if the underlying static analyzer misses intra- or inter-procedural flows, e.g., due to reflection, our interprocedural analysis in turn will miss action listeners.

Among the 437 non-ADF (i.e., true negative) apps, 384 have more than 1M installs, which again is a source of concern. Table 11 shows those 16 non-ADF apps with more than 10M installs. Neither the apps nor their websites offer account deletion, hence users cannot remove their account from the backend server. The Bleacher Report Live app provides an unusual option, to allow the app to sell user's personal information; the option is ON by default. However, the app does not allow the user to delete the account.

The column "#Non-ADF apps" in Table 7 shows the number of non-ADF apps in each LAI category. Email, password, location and phone number are the most frequent information collected by non-ADF apps – virtually all non-ADF apps collect such information.

TABLE 11: Non-ADF Apps with more than 10M Installs.

Package Name	#Ins-talls	Leftover Account Information
wp.wattpad	100M	Email, password, username, DOB, gender, location
com.zynga.words	50M	Email
com.xiaomi.hm.health	50M	Country, email, password
com.verizon.messaging.v	50M	Phone, country, email
com.neuralprisma	50M	Email, password
com.dataviz.docstogo	50M	First&last name, email
com.bleacherreport.andr.	10M	First&last name, username, phone, email
com.my.mail	10M	email, password, first&last name
com.period.tracker.lite	10M	Email, password
com.wsl.noom	10M	Email, password, unique program ID, gender, first name, age, height, weight, biograph
com.zynga.wwf.free	10M	Email
ru.yandex.mail	10M	Email, password
com.ryanair.cheapflts.	10M	Email, password, first&last name, DOB, nationality, country code, phone
com.delta.mobile.andr.	10M	First&last name, DOB, gender, username, password, email, phone, address, security questions 1&2, answers
com.cuvora.carinfo	10M	Phone#
com.br.netshoes	10M	Email, first&last name, DOB, CPF, ZIP, street, number, neighborhood, state, city, phone#, password

TABLE 12: Retention Period Analysis Results.

	True	FP	FN	Precision	Recall	F-measure
Apps (209)	34	3	3	91.9%	91.9%	91.9%

9.2.3. Retention Period Analyzer. Ground Truth was obtained via manual analysis on the 209 apps with ADF. We confirmed 34 apps with retention period. RetentionPeriodAnalyzer over-reported 3 apps and under-reported 3 apps, hence (Table 12) precision was 91.9%, recall was 91.9%, and the F-measure was 91.9%. The major reason for false negatives is that the account deletion button and retention period string appear in separate locations: one on the app website, the other in the APK file. The major reason for false positives was that the retention period strings did appear in XML files but were not displayed on app screens, e.g., the app Fever.

Retention period statistics and clusters. Table 13 shows that the maximum, minimum, average and median retention periods were 5 years, 30 minutes, 117.85 days, and 30 days respectively. The retention periods naturally fell into clusters (intervals), shown in Table 14. The most popular intervals were 7–30 days and 30–90 days.

TABLE 13: Retention Period Statistics.

Time (days)			
min	max	average	median
0.02	1,825	117.85	30

TABLE 14: Retention Period Intervals.

	Retention Period Clusters (days)			
	≤ 7	7–30	30–90	>90
#Apps	5	13	10	6

9.2.4. Leftover Account Cleaner. To study LeftoverAccountCleaner effectiveness, we created 10 test accounts on Google Play Store with varying numbers of uninstalled apps, then invoked the LeftoverAccountCleaner to perform automatic cleaning. An app is considered as cleaned up successfully if LeftoverAccountCleaner can automatically find and click the app's ADF button (or link) to delete the user account.

Table 15 shows our experimental results for the 10 users. The test users have uninstalled between 5 and 49 apps; these uninstalled apps are associated with leftover accounts which need to be cleaned up. We report the number of apps cleaned up successfully and unsuccessfully per each user. The experiment results show that LeftoverAccountCleaner failed to cleanup 12.65% of apps. Cleanup failed due to random advertising and CAPTCHA pop-ups. For example, during the screen auto-navigation stage, app Life360 (com.life360.android.safetymapd) shows a promotion ad pop-up; the ad asks users to upgrade to premium features. App com.discord has a CAPTCHA screen pop-up.

Note that, in the absence of LeftoverAccountCleaner, users would have to manually perform account cleanup, e.g., by reinstalling the app or going to the app's website *for up to 49 apps*, which is a substantial effort.

9.3. Efficiency

We now discuss the efficiency results for each tool. Statistics are shown in Table 16.

LeftoverAccountAnalyzer. The analyzer's median time was 162.68 seconds. The most time-consuming phase was the reinstall phase, which can take 30–60 seconds per app.

AccountDeletionAnalyzer. The median analysis time was 275.79 seconds per app and 3.03 seconds per website, respectively. The maximum time (16,407 seconds) was due to a lengthy Tesseract image-to-text conversion.

RetentionPeriodAnalyzer. The median analysis time was 259.03 seconds per mobile app and 2.96 seconds per website, respectively. The maximum time (10,311.26 seconds) was due to a lengthy Tesseract image-to-text conversion.

LeftoverAccountCleaner. The cleaning process' duration depends on (1) how many screens each app requires to navigate before reaching the account deletion button, and (2) how

TABLE 15: LAC Experimental Results.

Play Store Usernames (Anonymized)	Uninstalled Apps	Cleaned Accounts	
		Success	Failed
User1	9	8	1
User2	25	20	5
User3	43	37	6
User4	11	8	3
User5	27	26	1
User6	36	32	4
User7	5	5	0
User8	49	42	7
User9	21	19	2
User10	19	17	2

TABLE 16: Efficiency Results.

Analyzer/ Dataset	Time (seconds)			
	min	max	average	median
LeftoverAccountAnalyzer				
Mobile	30.44	1,185.96	189.78	162.68
AccountDeletionAnalyzer				
Mobile	0.57	16,407.67	369.44	275.78
Website	0.02	93.28	9.10	3.03
RetentionPeriodAnalyzer				
Mobile	0.80	10,311.26	345.92	259.03
Website	0.02	92.98	9.05	2.96
LeftoverAccountCleaner				
Account	10.53	1,104.22	266.51	231.29

many fields have to be filled-in on each screen. As shown in Table 16, cleaning up one app can take from 10.53 seconds to 1,104.22 seconds (median: 231.29 seconds).

To summarize, these results indicate that each tool in the chain is efficient, as is the overall approach.

10. Limitations

Our toolchain has four limitations. First, it cannot directly remove LAI from the backend database since the database is controlled by app developers/the app company. Second, the LeftoverAccountAnalyzer is not fully automated as some apps employ anti-automation mechanisms, e.g., CAPTCHA. Third, our NLP analyses can only handle English text. Fourth, the AccountDeletionAnalyzer cannot detect invisible buttons. All but the first limitation can be addressed with more engineering effort.

11. Related Work

There is a rich literature on privacy issues in mobile apps. However, we were not able to find any approach that focused on leftover accounts.

User control of personal data. The European Union's General Data Protection Regulation (GDPR) gives control of personal data back to the owners. Truong et al. [31] designed a GDPR-compliant personal data management platform. Vescovi et al. [32] developed a tool enabling people to control and share their personal data on mobile phones. Mun et al. [29] introduced a privacy architecture in which individuals retain ownership of their data.

Longitudinal privacy. Kröger et al. [25] performed a study on longitudinal privacy on mobile apps to examine how app vendors have complied with subject access requests over four years. Ayalon et al. [20] investigated the relation between information aging and its sharing preferences on Facebook. Mondal et al. [28] presented a study on understanding how users control the longitudinal exposure of their publicly shared social data.

Privacy leaks. Li et al. [26] proposed a static taint analyzer to detect privacy leaks among Android app components. Yang et al. [35] proposed an analysis framework to detect if sensitive user data is being transmitted out of an Android phone, whether users intend it or not. Gibler et al. [23] introduced a static analysis framework for automatically finding potential leaks of sensitive information in Android apps. Lin et al. [27] analyzed user mental models of mobile app privacy through crowdsourcing. Zuo et al. [37] investigated privacy leaks of Android apps, but in the Cloud. Zhang et al. [34] revealed that data persists on phones even after apps are uninstalled. Zimmeck et al. conducted an extensive privacy survey of Android apps [36]. Mylonas et al. proposed an approach for assessing the privacy risk of Android users based on the presence of specific permission combinations [30]. Wang et al. detected privacy leaks of user-entered data for an app and determined whether such leakage violate the app's privacy policy claims [33].

12. Conclusions

We expose and study the Leftover Account Information problem – information retained on app servers after an app is no longer used – which violates users' privacy. Our approach has four thrusts: leftover accounts after uninstalling apps, leftover accounts after deleting accounts, no account deletion functionality, and lack of retention period. Detecting such issues is complicated by several factors: lack of direct backend access requires LAI inference; both apps and their websites have to be analyzed; sophisticated NLP is required to extract and discern account deletion actions or retention policies; non-executable resources have to be connected to corresponding actions in executable bytecode. We address these challenges and develop four tools to detect LAI problems in Android apps. We ran the tools on a substantial corpus of popular Google Play apps, and revealed issues in hundreds of apps. Our study and tools can improve Android users' privacy by helping end-users, developers, and app marketplaces understand and mitigate the LAI problem.

Acknowledgments

We thank our anonymous shepherd and reviewers for their feedback. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1617584.

References

[1] "Zombie accounts: Should you be deleting old logins and forgotten profiles?" Oct 2018, https://globalnews.ca/news/4542449/zombie-accounts-should-you-be-deleting-old-logins-and-forgotten-profiles.
[2] "Analyzing sentence structure," Oct 2019, https://www.nltk.org/book/ch08.html.
[3] "Apktool - a tool for reverse engineering 3rd party, closed, binary android apps," Oct 2019, https://ibotpeaches.github.io/Apktool/.
[4] "Appium: Mobile app automation made awesome," Oct 2019, http://appium.io/.
[5] "Discord – privacy policy," Dec 2019, https://discordapp.com/privacy.
[6] "Don't just uninstall old apps—delete your accounts as well," Jul 2019, https://gizmodo.com/dont-just-uninstall-old-apps-delete-your-accounts-as-we-1836237449.
[7] "How to clear out your zombie apps and online accounts," Jul 2019, https://www.wired.com/story/delete-old-apps-accounts-online.
[8] "Infographic: Why users uninstall your mobile app?" Sep 2019, https://www.dotcominfoway.com/blog/infographic-why-users-uninstall-your-app/#gref.
[9] "Screaming frog seo spider tool & crawler software," Oct 2019, https://www.screamingfrog.co.uk/seo-spider/.
[10] "Semantic analysis (compilers) - wikipedia," Oct 2019, https://en.wikipedia.org/wiki/Semantic_analysis_(compilers).
[11] "tesseract-ocr/tesseract: Tesseract open source ocr engine (main repository)," Oct 2019, https://github.com/tesseract-ocr/tesseract.
[12] "8 tools to track android and ios app uninstalls," Oct 2020, https://appsamurai.com/8-tools-to-track-android-and-ios-app-uninstalls/.
[13] "Firebase summit 2020 livestream day 1," Oct 2020, https://youtu.be/lAbNYQPUGYM?t=1162.
[14] "Most popular installed backend software development kits (sdks) across android apps worldwide as of february 2020," Nov 2020, https://www.statista.com/statistics/1036080/leading-mobile-app-backend-sdks-android/.
[15] "Xposed framework," Oct 2020, https://www.xda-developers.com/xposed-framework-hub/.
[16] "mitmproxy," July 2021, https://mitmproxy.org/.
[17] AppBrain, "Number of android apps on google play," Oct 2020, https://www.appbrain.com/stats/number-of-android-apps.
[18] S. Arzt, S. Rasthofer, and E. Bodden, "The soot-based toolchain for analyzing android apps," ser. MOBILESoft '17. IEEE Press, 2017, p. 13–24. [Online]. Available: https://doi.org/10.1109/MOBILESoft.2017.2
[19] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299
[20] O. Ayalon and E. Toch, "Retrospective privacy: Managing longitudinal privacy in online social networks," in *Proceedings of the Ninth Symposium on Usable Privacy and Security*, ser. SOUPS '13. Association for Computing Machinery, 2013.
[21] A. Baidya, "Mobile app retention challenge: 75% users uninstall an app within 90 days," May 2016, https://dazeinfo.com/2016/05/19/mobile-app-retention-churn-rate-smartphone-users/.
[22] Emil Protalinski (VentureBeat), "Google updates firebase with new emulator and data analysis tools," Oct 2020, https://venturebeat.com/2020/10/27/google-updates-firebase-with-new-emulator-and-data-analysis-tools/.
[23] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale," in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, ser. TRUST'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 291–307. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30921-2_17
[24] Google, "Automatically collected events," Nov 2020, https://support.google.com/firebase/answer/6317485?hl=en.
[25] J. L. Kröger, J. Lindemann, and D. Herrmann, "How do app vendors respond to subject access requests? a longitudinal privacy study on ios and android apps," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ser. ARES '20. Association for Computing Machinery, 2020.
[26] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 280–291. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818791
[27] J. Lin, S. Amini, and J. I. Hong, "Expectation and purpose : Understanding users ' mental models of mobile app privacy through crowdsourcing," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. ACM, 2012, pp. 501–510.
[28] M. Mondal, J. Messias, S. Ghosh, K. Gummadi, and A. Kate, "Longitudinal privacy management in social media: The need for better controls," *IEEE Internet Computing*, pp. 1–1, 2017.
[29] M. Mun, S. Hao, N. Mishra, K. Shilton, J. Burke, D. Estrin, M. Hansen, and R. Govindan, "Personal data vaults: A locus of control for personal data streams," in *Proceedings of the 6th International COnference*, ser. Co-NEXT '10. Association for Computing Machinery, 2010.
[30] A. Mylonas, M. Theoharidou, and D. Gritzalis, "Assessing privacy risks in android: A user-centric approach," in *International Workshop on Risk Assessment and Risk-driven Testing*, 2013.
[31] N. B. Truong, K. Sun, G. M. Lee, and Y. Guo, "Gdpr-compliant personal data management: A blockchain-based solution," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1746–1761, 2020.
[32] M. Vescovi, C. Perentis, C. Leonardi, B. Lepri, and C. Moiso, "My data store: Toward user awareness and control on personal data," ser. UbiComp '14 Adjunct. Association for Computing Machinery, 2014, p. 179–182.
[33] X. Wang, X. Qin, M. B. Hosseini, R. Slavin, T. D. Breaux, and J. Niu, "Guileak: Tracing privacy policy claims on user input data for android applications," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 37–47.
[34] Z. Xiao, Y. Kailiang, A. Yousra, Q. Zhenshen, and D. Wenliang, "Life after app uninstallation: Are the data still alive? data residue attacks on android," in *Proceedings of the 23rd Network & Distributed System Security Symposium*, 2016, pp. 1–15.
[35] Z. Yang, M. Yang, and X. S. Wang, "Appintent : Analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1043–1054.
[36] S. Zimmeck, P. Story, A. Ravichander, D. Smullen, Z. Wang, J. Reidenberg, N. C. Russell, and N. Sadeh, "MAPS: Scaling privacy compliance analysis to a million apps," in *19th Privacy Enhancing Technologies Symposium (PETS 2019)*, vol. 3. Stockholm, Sweden: Sciendo, July 2019, pp. 66–86.
[37] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *Proceedings of the 40th IEEE Symposium on Security and Privacy*, 2019.

13. Appendix

13.1. Tool Chain Architecture

Figure 7 shows the components of our tool chain (in the center), the third-party frameworks and tools we leveraged, and the flow diagram between these components.

We leverage (or build on top of) various kinds of tools, for static analysis, dynamic analysis, automatic testing and website crawling, NLP and OCR.

Our *static analyses* are built on top of Soot/Flowdroid. We extended Soot/Flowdroid to permit a finer data-flow/def-use-chains analysis (between arbitrary pairs of statements, rather than just between predefined sources and sinks).

Dynamic analysis tools include the third-party Appium (for automated testing) and our own LAI Monitor (for intercepting and logging AM operations). We developed the LAI Monitor on top of the Xposed framework. Our LAI Monitor

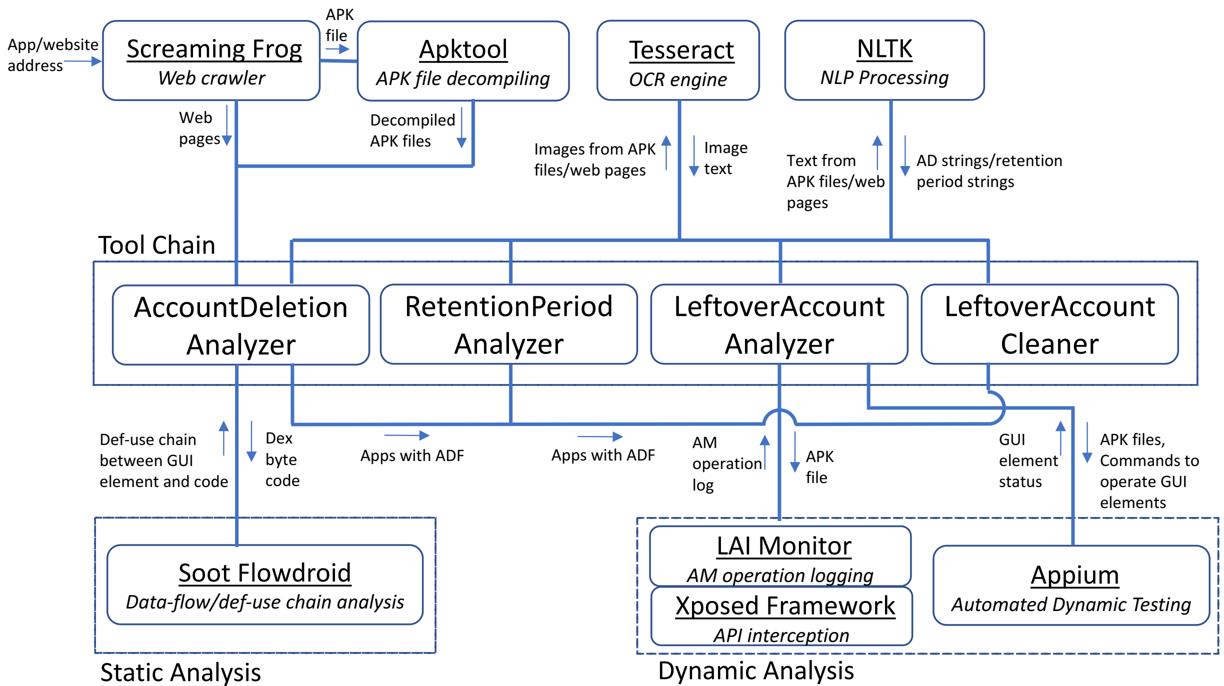


Fig. 7: Tool Chain Architecture.

is configurable to intercept various API calls, including the Android, Firebase, or Java APIs. Note that when the user creates or deletes an account, Android/Java/Firebase APIs will be called; our LAI Monitor logs the AM operation before the messages are sent to the backend server. The LAI Monitor implementation and manual are available on GitHub.[6]

For *NLP* we leverage the NLTK library to identify AD strings and retention period strings. For *OCR*, Tesseract is used to extract text from image assets, e.g., figures embedded with the app or on the website. For web crawling, Screaming Frog is used to automatically download APK files and crawl web pages.

13.2. Trees for AD String Detection

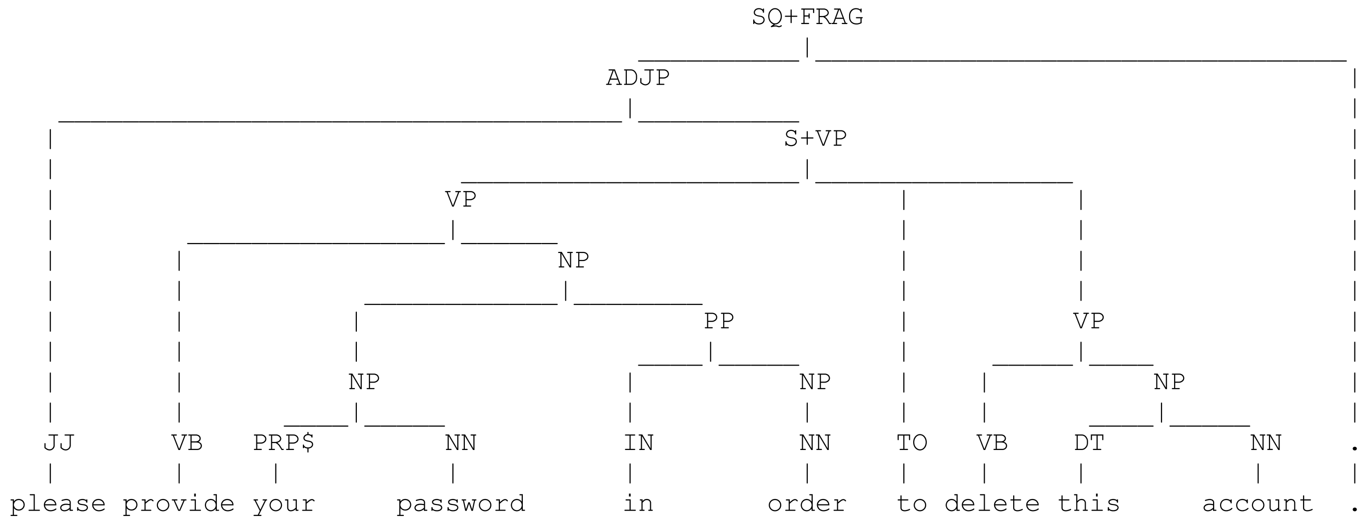
To illustrate the grammar, we use phrase structure trees [2] to show the semantic structure [10] of a sentence. Each node in the tree (including the words) is called a constituent. Figure 8 shows two example apps' phrase structure trees.

In Figure 8-top, the Fitbit app's tree has a $Verbphrase$ subtree whose verb is an AD verb. The subtree contains a $Nounphrase$ subtree whose $Noun$ is an AD noun. As this string is in the language induced by the grammar, we deem it an *AD string*.

In contrast, for the Zomato app (Figure 8-bottom) the phrase structure tree contains AD verbs and AD nouns, but it does not conform to the grammar and is labeled as a *non-AD string* because the succeeding subtree of the $Verbphrase$ is a propositional phrase tree (as opposed to $Nounphrase$).

6. https://github.com/LeftoverAccountInformation/LAI/tree/master/LeftoverAccountAnalyzer/LaiMonitor

Fitbit: **AD**



Zomato: **Not AD**

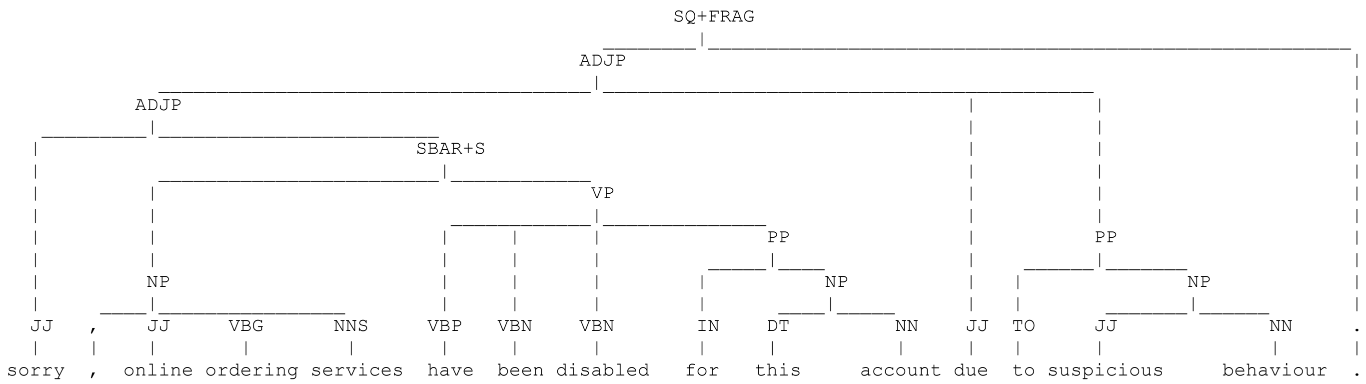


Fig. 8: Phrase Structure Trees Discerning AD from Not AD. Abbreviations: S(Simple declarative clause), SQ+FRAG(Sentence Fragment), ADJP(Adjective Phrase), SBAR(Clause introduced by a subordinating conjunction), VP(Verb Phrase), NP(Noun Phrase), PP(Prepositional Phrase), JJ(Adjective), VB(Verb, base form), PRP$(Possessive pronoun), NN(Noun), NNS(Noun, plural), IN(Preposition or subordinating conjunction), TO(to), DT(Determiner), VBG(Verb, gerund or present participle), VBN(Verb, past participle), VBP(Verb, non-3rd person singular present)