2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

Mining Sandboxes

Konrad Jamrozik · Philipp von Styp-Rekowsky · Andreas Zeller
Center for IT-Security, Privacy and Accountability (CISPA), Saarbrücken, Germany
{jamrozik,styp-rekowsky,zeller}@cs.uni-saarland.de

ABSTRACT

We present *sandbox mining,* a technique to confine an application to resources accessed during automatic testing. Sandbox mining first explores software behavior by means of *automatic test generation,* and extracts the set of resources accessed during these tests. This set is then used as a *sandbox,* blocking access to resources not used during testing. The mined sandbox thus protects against *behavior changes* such as the activation of latent malware, infections, targeted attacks, or malicious updates.

The use of test generation makes sandbox mining a fully automatic process that can be run by vendors and end users alike. Our BOXMATE prototype requires less than one hour to extract a sandbox from an Android app, with few to no confirmations required for frequently used functionality.

1. INTRODUCTION

How can I protect my computer from malicious programs? One way is to place the program in a *sandbox,* restraining its access to potentially sensitive resources and services. On the Android platform, for instance, developers have to declare that an application (henceforth referred to as an app) needs access to specific resources. The popular SNAPCHAT picture messaging application, for instance, requires permissions to access the Internet, the camera, and the user's contacts. To install the app the user has to grant such permissions. If an application fails to declare a permission, the operating system denies access to the respective resource; if the SNAPCHAT app attempted to access e-mail or text messages, the respective API call would be denied by the Android system.

While such permissions are transparent to users, they may be too *coarse-grained* to prevent misuse. For instance, SNAPCHAT offers a feature to find friends on SNAPCHAT based on their phone number. To do this, SNAPCHAT accesses the phone numbers of the user's contacts, and sends them to the SNAPCHAT servers. The permission given by the Android sandbox allows SNAPCHAT to do much more than that, namely unlimited access to *all* contacts at *any* time. An attacker thus could inject malware into a SNAPCHAT binary that compromises all contact details; the permissions could stay unchanged. The issue could be addressed by *tightening* the sandbox—for instance, by constraining the conditions under which the app can send the message. But then, someone has to specify and validate these rules—and repeat this with each change to the app, as a sandbox that is too tight could disable important functionality.

ICSE '16, May 14 - 22, 2016, Austin, TX, USA

ISBN 978-1-4503-3900-1/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2884781.2884782

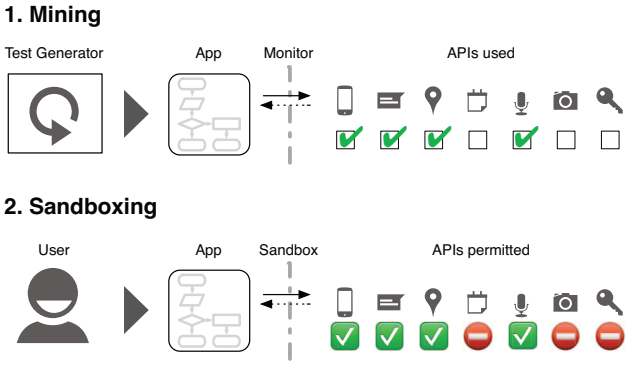

Figure 1: Sandbox mining in a nutshell. The mining phase automatically generates tests for an application, monitoring the accessed APIs and resources. These make up the *sandbox* for the app, which later prohibits access to resources not accessed during testing.

In this paper, we present *sandbox mining,* a technique to *automatically extract sandbox rules* from a given program. The core idea, illustrated in Figure 1, brings together two techniques, namely *test generation* and *enforcement*, in a principle called *test complement exclusion*—disallowing behavior not seen during testing:

Mining. In the first phase, we *mine* the rules that will make the sandbox. We use an *automatic test generator* to systematically explore program behavior, monitoring all accesses to sensitive resources.

Sandboxing. In the second phase, we assume that *resources not accessed during testing should not be accessed in production either.* Consequently, if the app (unexpectedly) requires access to a new resource, the sandbox will prohibit access, or put the request on hold until the user explicitly allows it.

To illustrate how test complement exclusion works in practice, let us mine a sandbox from our SNAPCHAT example application. During systematic GUI testing, the mining phase determines that SNAPCHAT indeed requires access to the camera, location, Internet, and so on. We associate these accesses with the event that triggers them—that is, the individual GUI elements. Thus, we would find that SNAPCHAT accesses contacts only when the user presses the "Find friends" GUI button; and it only accesses the friends' phone numbers. Likewise, accessing the microphone or the location would only take place when a message is actually sent.

The resulting sandbox is *much more fine-grained* than the original Android sandbox, and easily prevents a number of otherwise permitted attack schemes. Compromising all contact data, sending text messages in the background, continuously monitoring the audio or the current location, would all be disallowed, simply because this behavior is not what we find during testing.

Even more important, though, is that the sandbox also protects the user against *unexpected behavior changes.* Assume an app like SNAPCHAT was malicious in the first place, and placed in an app store. Then, the attacker would face a dilemma. If the app accesses all contacts right after the start, this would be detected in the mining phase, and thus made explicit as a sandbox rule permitting behavior; such a rule ("This app reads all contact details in the background") could raise suspicions even with non-expert users, because there is no apparent functionality in SNAPCHAT that requires this. If, however, the app stayed benign during mining, it would be disallowed from accessing contact details in production, except for phone numbers during the "Find friends" functionality.

To the best of our knowledge, ours is the first approach to *leverage test generation to automatically extract sandbox rules from general-purpose applications.* Sandbox mining has a number of compelling features:

Preventing behavior changes. The mined sandbox detects behavior not seen during mining, reducing the attack surface for infections as well as for *latent* malicious behavior that otherwise would activate later.

Fully automatic. As soon as an interface for automatic test generation is available, such as a GUI, sandbox mining also becomes fully automatic. Developers can easily mine and remine sandboxes at any time.

No training in production. In contrast to anomaly detection systems, we need no training in production, as the "normal" behavior would already be explored during testing.

Detailed analysis. Mined sandboxes provide a much finer level of detail than what would normally be specified or documented in practice. As they refer to user resources and user actions, they are readable and understandable even by non-experts.

Adverse and obscure code. In contrast to static code analysis, test generation and monitoring are neither challenged by large programs nor thwarted by code that would be deobfuscated, decrypted, interpreted, or downloaded at runtime only.

Guarantees from testing. The key issue with testing is that it is incomplete by construction. We turn it to our advantage by considering the tested behavior a safe subset of all possible app behaviors, guaranteeing the user will be explicitly asked to allow behaviors not seen during testing.

Certification. Anyone can mine a sandbox for a given app and compare its rules against the sandboxes provided by others, or those of previous versions.

All of this, however, depends on a number of assumptions that can only be assessed in a practical setting. Our BOXMATE tool[1] implements sandbox mining for the Android platform in a user-friendly package, consisting of the DROIDMATE test generator and the BOXIFY [5] approach to privacy enforcement. After discussing related work in Section 2, we address three key questions:

Q1 *Can test generators sufficiently cover behavior?* If some resource R is not accessed during mining, later non-malicious access to R would require user confirmation—the sandbox is *too tight.* We run the DROIDMATE test generator on a set of Android apps, checking API coverage (Section 3) and check when the sandbox would trigger which alarms (Section 4).

Q2 *Can we sufficiently reduce the attack surface?* If the rules we mine are too general, there might still be too many ways for applications to behave maliciously—the sandbox is *too coarse.* To this end, we associate resource access with the GUI elements that trigger them (Section 5), further reducing the attack surface.

Q3 *Can sandbox rules help experts to assess behavior?* If the analyzed app is overtly malicious, the mined sandbox will not prevent this. Section 6 shows how mined sandbox rules help in assessing and comparing behavior in the first place, reducing the risk of missing an attack.

After discussing threats to validity and limitations (Section 7), Section 8 closes with conclusion and future work.

2. BACKGROUND

2.1 Sandboxing

The idea of restricting program operation to only the information and resources necessary to complete its operation goes back to the 1970s. As *principle of least privilege* [34], it has influenced the design of computer systems, operating systems, and information systems to improve stability, safety, security, and privacy. On the Android platform, least privilege is realized through *sandboxing:* First, no application can access the data of other applications. Second, access to shared user resources (such as location, contacts, etc.) is available through dedicated APIs only, which are guarded by *permissions*. Each application declares the permissions it needs; the operating system blocks access to other APIs and resources.

In a landmark paper, Felt et al. [17] systematically tested Android APIs to check which permissions would refer to which API. Besides producing a map between APIs and permissions, they also found that 33% of Android apps investigated were *overprivileged*, that is, they requested more permissions than their APIs would actually require. PSCOUT [4] uses a combination of static analysis and fuzz testing to extend this mapping to *undocumented APIs*, and found that 22% of permission requests were unnecessary if app developers confined themselves to documented APIs.

These Android permissions have to be acknowledged by a user upon app installation; the Google play store also lists each app with the requested permissions. However, in a survey of 308 Android users, Felt et al. [18] found that only 17% paid attention to permissions during installation, and only 3% of all respondents could correctly answer three questions regarding permissions. The latest Android version 6 therefore adopts the iOS security model, asking for permission interactively and in context at the very moment the app accesses a sensitive resource. Most permissions such as Internet access are granted by default, though, and the set of confirmations is limited to the most sensitive resources only.

The Android permission model is coarse-grained.

2.2 Analyzing Apps

In contrast to *specified* rules and permissions, the alternative of *extracting* these from an existing system has always been compelling. In a short paper comparing the permission systems of mobile platforms [3], Au et al. call for "a tool that can automatically determine the permissions an application needs." This question generalizes into "What does an application do?", which is the general problem of *program analysis.*

[1] BOXMATE = Sandbox mining, analysis, testing and enforcement

Program analysis falls into two categories: *static* analysis of program code and *dynamic* analysis of executions. Static code analysis sets an *upper bound* to what a program can do: If static analysis determines some behavior is impossible, it can be safely excluded. Tools like CHEX [26] and FLOWDROID [2] check Android apps for information flow between sensitive sources and sinks. The COPES framework [8] uses static analysis to eliminate unneeded permissions for a given Android app.

The challenge of *static analysis* is *overapproximation:* The analysis must often assume that more behaviors are possible than actually would be. The analysis is undecidable in all generality due to the halting problem. Also, static analysis is challenged by code that is decrypted, interpreted, or downloaded at runtime only—techniques used by benign and malicious Android apps alike. *If* static analysis can safely determine that some behavior is impossible, though, the behavior no longer needs to be checked at runtime.

Static analysis produces overapproximation.

Dynamic analysis works on actual *executions,* and thus is not limited by code properties. In terms of program behavior, it sets a *lower bound:* Any (benign) behavior seen in past executions should be allowed in the future, too. Consequently, given a set of executions, one can *learn* program behavior from these and infer security policies. However, obfuscated or encrypted code makes it harder to infer the behavior's intent. In their seminal 1996 paper [19], Forrest et al. learned "normal" behavior as short-range correlations in the system calls of a UNIX process, and were successfully able to detect common intrusions on the *sendmail* and *lpr* programs. Since then, a number of techniques have been used for automatic anomaly detection; Chandola et al. [13] provide a detailed survey. Most related to BOXMATE is the work of Provos [32], learning and enforcing policies for system calls on UNIX systems.

Since Android programs come in interpretable byte code, the platform offers several opportunities to monitor dynamic behavior, including system calls (*AASandbox* [11]), data flow (TAINTDROID [14]), traces (CROWDROID [12]), or CPU and network activity (ANDROMALY [35]); all these platforms can be used both to *monitor* application behavior (and report results to the user) as well as to *detect* malicious behavior (as a violation of explicit rules or as determined by a trained classifier). Neuner et al. [31] as well as Lindorfer et al. [25] provide a comprehensive survey of trends and available techniques.

Dynamic behavior can also be abstracted and summarized using *internal state,* following the pioneering work of Ernst et al. on dynamic invariants [16] and the suggestion of Engler et al. that deviations in behavior would help in inferring errors [15]. Baliga et al., for instance, would learn kernel data structure invariants to detect rootkits [7]. As they refer to internal state, the diagnostics of these approaches cater more to developers than to users or administrators, though; however, they also share the general idea of learning "normal" behavior to detect "abnormal" behaviors.

The joint problem of all these approaches is the fundamental limitation of dynamic analysis, namely *incompleteness:* If some behavior has not been observed so far, there is *no guarantee that it may not occur in the future.* Given the high cost of false alarms, this implies that a sufficiently large set of executions must be available that covers known behaviors. Such a set can either come from *tests* (which then typically would be written or conducted at substantial effort), or from *production* (which then requires a training phase, possibly involving classification by humans). In the domain of network intrusion detection, the large variation of "benign" traffic in operational "real world" settings is seen as a prime reason why machine learning is rarely employed in practice [37].

Dynamic analysis requires sufficiently many "normal" executions to be trained with.

2.3 Test Generation

Rather than write tests or collect executions during production, one can also *generate* them. In the security domain, the main purpose of such generated executions is to find bugs. Introduced by Miller et al. [29], *fuzz testing* automatically exercises sensitive tools and APIs with random inputs; no interaction or annotation is required. Today, fuzz testing is one of the prime methods to find vulnerabilities: The Microsoft SAGE fuzzing tool [20], for instance, "has saved millions of dollars to Microsoft, as well as to the world in time and energy, by avoiding expensive security patches to more than 1 billion PCs." [21].

For the Android platform, recent years have seen a raise of powerful test generators exercising Android apps. MONKEY [30] is a simple fuzz tester, generating random streams of user events such as clicks, touches, or gestures; although typically used as robustness tester, it has been used to find GUI bugs [24] and security bugs [28]. While MONKEY generates pure random events, the DYNODROID tool [27] focuses on those events handled by an app, getting higher coverage while needing only 1/20 of the events. Given an app, all these tools run fully automatically; no model, app code, or annotation is required. Other recent Android test generators like PUMA [23] or ANDLANTIS [10] achieve high levels of robustness, while BRAHMASTRA [9] is good at covering 3rd party components.

All these testing tools still share the fundamental limitation of execution analysis: If a behavior has not been found during testing, there is no guarantee it will not occur in the future. Attackers can easily exploit this by making malicious behavior *latent:* For instance, our malicious SNAPCHAT variant would start sending malicious text messages only after some time, or in a specific network, or when no dynamic analysis tool is run, each of which would defeat observation during testing.

Testing cannot guarantee the absence of malicious behavior.

2.4 Consequences

Program analysis, sandboxing, and test generation are all mature technologies that are sufficiently robust to be applied on a large scale. However, each of them has fundamental limitations—sandboxes need rules, dynamic analysis needs executions, and testing does not provide guarantees. Combining the three, however, not only mitigates these weaknesses—it even turns them into a strength. The argument, first presented in a keynote at the ICPC 2015 conference [38], is as follows:

With modern test generators, we can generate as many executions as needed. These executions can feed dynamic analysis, providing and summarizing insights into what happens in them. By construction, these insights are incomplete, and other (in particular malicious) behavior is still possible. The key idea of this paper is to *turn the incompleteness of dynamic analysis into a guarantee*—namely by having a sandbox enforce that anything not seen yet will not happen. To the best of our knowledge, this is the first work bringing together test generation, dynamic analysis, and sandboxing; it is their combined strength we explore in this paper.

3. GENERATING APP TESTS

Let us now detail how DROIDMATE, the test generator of BOXMATE, operates. Conceptually, DROIDMATE generates tests by *exploring* the *Application under Test* (*AuT*), that is, by interacting at

runtime with its GUI elements (called *views* in Android) and reasoning about the *AuT* behavior to influence further GUI interaction.

DROIDMATE installs on an Android device an .apk file containing the *AuT* and then launches the *AuT*'s main activity.[2] During start, and then again after each generated interaction, DROIDMATE monitors which sensitive Android APIs and user resources the *AuT* accesses. As the exploration progresses, all the observed and monitored behavior of the *AuT* is being used to decide which GUI element to interact with next or if to terminate the exploration. The data from the exploration is sufficient to replay the test, either manually or automatically.

The exploration takes place in a loop between an *exploration strategy* and an *exploration driver*. The exploration strategy algorithm is given in Algorithm 1. It operates on a high abstraction level, taking as input the *GUI state* and returning an *exploration action*. The GUI state contains an abstract representation of the GUI, hiding all the implementation details irrelevant for deciding what to explore next. The exploration action in turn is an abstract representation of a possibly multi-step operation on the Android device like *click, long-click, reset* or *terminate.* The exploration driver then translates this abstract representation into actual operations on the device, executes them, reads the resulting GUI state and API calls logs, and returns control to the exploration strategy.

The actual exploration strategy currently implemented in DROIDMATE is inspired by DYNODROID [27]. The key idea is to *interact* with views (GUI elements) randomly, but give precedence to views that have been interacted with the least amount of times so far. If multiple views have been interacted with minimal amount of times, we pick one randomly. A view interaction is either a click or a long click (2 seconds). Interaction can happen only with views that are *enabled* as well as *clickable*, *long-clickable*, or *checkable*.

Each view is considered unique in its given *context*—that is, within the set of views that can be interacted with and appear on the same screen. Thus, if a view appears in different contexts (i.e., surrounded by different GUI elements), it will be explored again in each of them. Contexts are different if they differ by at least one view. A view can differ by its fully qualified *class name,* its *resource ID* (if any), its *content description* (if any) and the rectangle describing its location on the screen. It can also differ by its *label,* unless the view's class has `Switch` or `Toggle` in its name.

Every 30 interactions, DROIDMATE restarts the *AuT*. We thus avoid getting stuck in abnormal situations such as no views being available for interaction, the app having crashed, or another app having been launched. A view that led to a reset gets black-listed and will not be interacted with again. The exploration terminates when the configured time limit is reached or when there are no views that can be interacted with after two resets in a row.

3.1 Distinguishing Resources

While running, DROIDMATE monitors *sensitive Android API calls*, using the monitoring techniques discussed in Section 4. An API is *sensitive* if it is governed by a permission. We use the set of sensitive APIs used in the APPGUARD privacy-control framework [6]. This set of 97 APIs focuses on crucial privacy-related resources an average user should be concerned about.[3]

For each call of a monitored API, DROIDMATE records

1. The fully qualified name of the API called, including class and method name and parameter and return types;
2. The thread ID and the entire thread call stack trace of the API call (starting at *Thread.run()* or Dalvik's native *main()*);
3. Properties of the triggering view like displayed text, associated resource ID, screen bounds, etc.

[2]If the *AuT* accesses an external account, such as SNAPCHAT, its login and password must be provided.

[3]The full API list is provided in the linked experimental data package (Section 8).

Algorithm 1 Exploration strategy.

Require: GUI State S
Ensure: Exploration action A

```
1: procedure DECIDE(S)
2:     if TERMINATE(S) then
3:         A ← terminate exploration
4:     else
5:         if RESET(S) then
6:             A ← reset exploration
7:         else
8:             A ←EXPLOREFORWARD(S)
9:         end if
10:    end if
11:    UPDATEINTERNALSTATE(S,A)
12:    return A;
13: end procedure
14:
15: procedure EXPLOREFORWARD(S)
16:    C ← view context of S
17:    VS ← views in C with minimal number of interactions so
       far
18:    V ← pick at random from VS
19:    A ← choose interaction action with V
20:    UPDATEKNOWNVIEWCONTEXTS(C)
21:    UPDATEINTERACTIONSCOUNT(V,C)
22:    return A
23: end procedure
```

As most Android resources are uniquely identified by their specific set of APIs, we can ignore parameter values in most cases: they determine irrelevant details, e.g. a call to *LocationManager .requestLocationUpdates(listener)* determines which listener to inform when a location has changed. Yet we are interested only if appropriate call to *LocationManager* was made at all.

However, one set of Android API methods heavily depend on the parameter values to identify the correct resource accessed and therefore get special treatment. These are *ContentResolvers*—that is, database equivalents frequently used in Android. Knowing only that *ContentResolver.query()* was called is not enough, as the query may relate to all kinds of sensitive resources. For *ContentResolver* calls, DROIDMATE therefore also monitors the URI identifying the exact database, e.g. *content://com.android.contacts/data/phones*. Sometimes, URIs end with the numeric identifier of particular instance of the resource being accessed: we consider all API calls differing only by this number as equivalent.

3.2 Mining SNAPCHAT

As an example of how DROIDMATE explores application behavior, let us again consider the SNAPCHAT application. Figure 2 lists the number of unique APIs discovered during testing; the actual APIs (in order of discovery) are listed in Figure 3, including the identifiers of the GUI elements that triggered them:

API 1 After a click on the *login_button* on the start view, SNAPCHAT opens a socket (API 1) which allows establishing a connection to a HTTP server. It also opens the camera (API 2), queries the current location (API 3) and accesses account info via a URL connection (API 4).

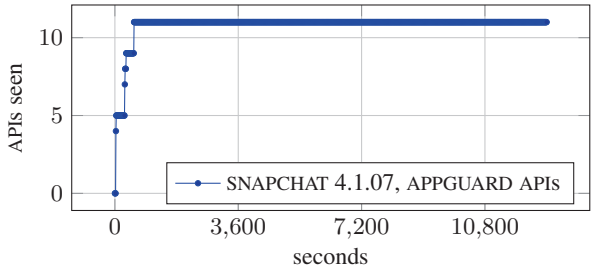

Figure 2: DROIDMATE per-app API saturation. After 10 minutes (600 seconds), DROIDMATE has discovered 11 sensitive APIs used by SNAPCHAT.

API 5 Taking a picture (*camera_take_snap_button*) starts monitoring the current location.

API 6 Recording a video sets the video and audio sources for recording, initializing the media recorder.

API 8 Later, DROIDMATE finds the SNAPCHAT "My friends" button (the unlabeled element), which requires accesses to the image library.

API 9 SNAPCHAT allows for finding friends based on their phone number, requiring access to contacts.

API 10 Saving a picture stores it to a database.

API 11 Previewing a snap deletes it after the preview is done.

```
[Button com.snapchat.android:id/login_button]
   1  java.net.Socket: void <init>
   2  android.hardware.Camera.open()
   3  android.location.LocationManager.getLastKnownLocation()
   4  java.net.URLConnection openConnection()
[Button com.snapchat.android:id/camera_take_snap_button]
   5  android.location.LocationManager.isProviderEnabled()
[Button com.snapchat.android:id/camera_take_snap_button
  (long-click)]
   6  android.media.MediaRecorder.setAudioSource()
   7  android.media.MediaRecorder.setVideoSource()
[unlabeled GUI element]
   8  android.content.ContentResolver.query()
        uri = content://media/external/images/media
[Button com.snapchat.android:id/contacts_permission_button]
   9  android.content.ContentResolver.query()
        uri = content://com.android.contacts/data/phones
[ImageButton com.snapchat.android:id/picture_save_pic]
  10  android.content.ContentResolver.insert()
        uri = content://media/external/images/media
[RelativeLayout com.snapchat.android:id/
  snap_preview_relative_layout]
  11  android.content.ContentResolver.delete()
        uri = content://media/external/images/media/<number>
```

Figure 3: The 11 SNAPCHAT calls to sensitive APIs discovered by DROIDMATE, and the events (in `[]`) that first trigger them.

These APIs characterize the resources that SNAPCHAT accesses—or more precisely, the resources it accessed in our DROIDMATE run. So are these 11 APIs an exhaustive list? This is the problem of testing, which does not give guarantee of whether all has been seen; and this is why we use sandboxing, to prevent yet unseen, potentially malicious behavior.

4. MONITORING AND ENFORCING

Besides a test generator, the second component of BOXMATE is the sandbox mechanism itself, monitoring (and possibly preventing) program behavior. Just as with test generation, we wanted a technique that allows any *user* to sandbox *any* application on an

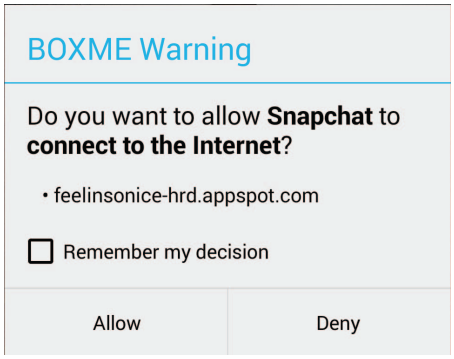

Figure 4: The BOXMATE sandbox in action. Calling a sensitive API not seen during mining requires confirmation by the user. To facilitate readability, API names are automatically mapped into the respective Android permissions, which are then shown in user-readable form.

unmodified Android device. To this end, we leveraged the BOXIFY tool by Backes et al. [5].

4.1 Monitoring in a Nutshell

The BOXMATE monitoring component uses the BOXIFY framework for fine-grained policy enforcement [5]. BOXIFY is a novel approach for Android application sandboxing, which provides tamper-protected reference monitoring for stock Android without the need for root privileges. BOXIFY uses app virtualization and process-based privilege separation to encapsulate untrusted applications in a restricted execution environment within the context of another, trusted sandbox application. To establish a restricted execution environment, BOXIFY leverages Android's *isolated process* feature, which allows apps to completely de-privilege selected components. By loading untrusted apps into de-privileged, isolated processes, BOXIFY avoids modifying apps and provides strong security guarantees. Sensitive I/O operations are relayed through a separate, privileged broker process monitoring and enforcing policies.

The BOXMATE sandbox works in two modes. During *mining,* it records and distinguishes all calls to sensitive APIs; as discussed in Section 3.1, this recording includes the current call stack, the thread ID as well as security-relevant parameter values. During *enforcement,* it checks whether the API call is allowed by the mined sandbox rules; if not, it can either have the call return a *mock object* containing fake data, or *flag* the call, asking the user for permission, naming the API and possible relevant arguments (Figure 4). If the user declines permission, the call is denied. Being based on BOXIFY, only calls to sensitive methods incur an overhead of 1–12% per call [5], resulting in practically no runtime performance overhead.

As an example of how the BOXMATE sandbox operates, again consider the SNAPCHAT saturation curve in Figure 2. Any sensitive API not accessed during testing–that is, any call to an API not listed in Figure 3—would be flagged by the BOXMATE sandbox. Note how the BOXMATE sandbox is already much more fine-grained than, say, the standard Android permission model. In the Android permission model, for instance, SNAPCHAT would simply get arbitrary access to all contacts. In the BOXMATE model, though, SNAPCHAT is only allowed to read contact phone numbers; any other information is neither accessed nor changed. These are important features to know, and possibly to enforce, too.

4.2 Evaluation

Since any sensitive API not explored during testing implies a potential false alarm during production, we *evaluate the risk of false*

Table 1: Evaluation Subjects. Open `https://play.google.com/store/apps/details?id=`⟨Identifier⟩ for details.

Name	Version	Category	Rank	Identifier (links to Web page)
Adobe Reader	11.1.3	Productivity	1	com.adobe.reader
AntiVirus Security – FREE	3.6	Communication	5	com.antivirus
Barcode & QR Scanner barcoo	3.6	Shopping	6	de.barcoo.android
CleanMaster – Free Optimizer	5.1.0	Tool	1	com.cleanmaster.security
Currency converter	1.02	Finance	9	com.frank_weber.forex2
eBay	2.5.0.31	Shopping	1	com.ebay.mobile
ES Task Manager(Task Killer)	1.4.2	Business	10	com.estrongs.android.taskmanager
Expense Manager	2.2.3	Finance	24	at.markushi.expensemanager
File Manager (Explorer)	1.16.7	Business	1	com.rhmsoft.fm
Firefox Browser for Android	28.0.1	Communication	7	org.mozilla.firefox
Job Search	2.3	Business	6	com.indeed.android.jobsearch
PicsArt – Photo Studio	4.1.1	Photography	1	com.picsart.studio
Snapchat	4.1.07	Social	4	com.snapchat.android

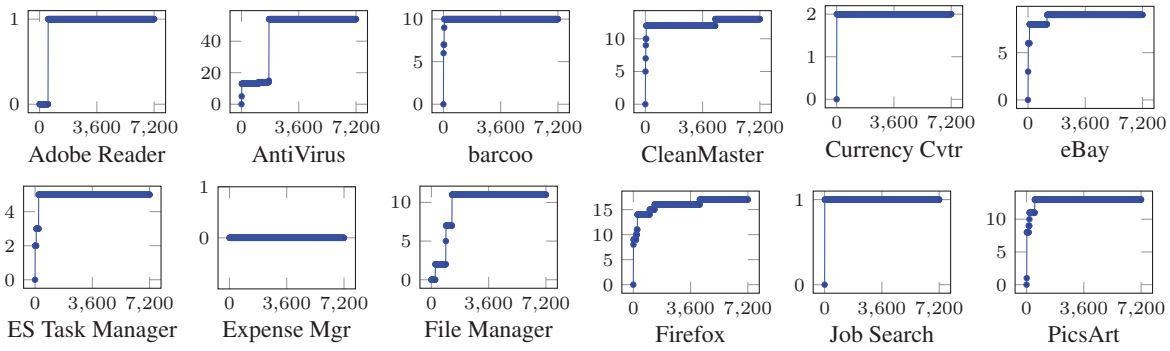

Figure 5: Per-app saturation for the apps in Table 1. As in Figure 2, the y axis is APIs seen, the x axis is seconds spent.

alarms: How likely is it that sandbox mining misses functionality, and how frequently will users thus encounter false alarms? We address this issue from two angles: We evaluate how quickly the set of APIs is saturated (Section 4.2.1) and we check BOXMATE against 18 use cases reflecting typical app usage (Section 4.2.2).

4.2.1 Exploration

While a finer-grained access model reduces the attack surface, it also brings the risk of *false alarms.* In Figure 2, just 10 minutes of mining is enough. The question is whether other apps can be also quickly mined, covering all the frequently used functions. To this end, we computed the same API saturation for twelve more apps (Table 1) from the top downloads of the Google Play Store. Figure 5 shows the respective API saturation charts; these are the same charts as we have already seen for SNAPCHAT in Figure 2. We see that ten charts "flatten" before one hour mark and the remaining two before two hours.

Automatic test generation can quickly cover resource usage.

4.2.2 Use Cases

We now know DROIDMATE stops discovering new calls to sensitive APIs before two hours pass. But does this mean that the most important functionality is actually found at all? To answer this question, we defined *use cases* for each of the analyzed apps, reflecting their most important usages. We derived the use cases from the app's main purpose as stated in its description—viewing a PDF document with Adobe Reader, scanning the system with AntiVirus Security, sending a picture with SNAPCHAT, and so on. Table 2 provides a full list of the defined use cases.

We implemented all these use cases as *automated test cases,* allowing for easy assessment and replication of our experiments. On average, implementing a single use case and having it replay reliably took us 2–3 hours of work. This perhaps surprisingly high implementation effort was due to inaccuracies in the *uiautomator* framework as well as the general difficulty of hand-scripting user interactions (which in turn may further motivate the use of automated exploration frameworks such as DROIDMATE).

After having BOXMATE extract the sandbox for a given app, the central question for the evaluation would be whether (and if so, how) these use cases would be impacted by the sandbox.

The "app" column in Table 2 summarizes the number of confirmations a user has to provide in the APPGUARD APIs set. The PicsArt "Apply effect" accesses an existing photo from SD card, which was not found during testing. The eBay "Find by Search" use case requires login credentials, while we explicitly didn't gave them to DROIDMATE forcing it to explore only the functionality available without logging. The use case in turn explores GUI parts available only after logging, causing the need for confirmation. This answers **Q1:** Only in 2 out of 18 use cases, each encompassing up to dozens of sensitive API calls, would a user need to confirm API access. This is actually fewer confirmations than in Android 6, where first access to every permission group has to be explicitly confirmed once per app [1].

Mined sandboxes require fewer user confirmations than standard OS security facilities.

[4]Despite our best efforts, neither we nor DROIDMATE could get barcoo 3.6 to use the camera and scan something on our devices.

Table 2: Use cases. Confirmations required with APPGUARD API calls ("app" column); and (*event*, *api*) pairs ("event" column).

App	Use Case	Functions	Confirmations per: app	event
Adobe Reader	View Document	What's New, Help, Open first document	–	–
AntiVirus	Scan	Activate, Scan now, View scan results	–	–
barcoo	Search for product	Search "pillow" in search box, View results[4]	–	1
CleanMaster	Scan	Scan system, Resolve all, Report	–	3
Currency Cvtr	Convert currency	Enter "159", Swap currencies	–	–
eBay	Find by search	Accept terms, Sign in, Search "pillow", View first search result	1	1
ES Task Mgr	Kill task	Kill first listed task	–	–
Expense Mgr	Add and edit expense	Add an expense of $15.80 for "Pills" in Category "Health"	–	–
	Delete expense	Open history, Delete first entry	–	–
	View and set budget	Set a total budget of $7.00 in the "other" category	–	–
File Manager	View and create dir	View directories, create new directory "temp_utc"	–	–
Firefox	Open URL	Go to "google.com"	–	–
Job Search	Search for job	Search a job for "sales" in "New York, NY", Select first result	–	–
PicsArt	Apply effect	Apply "twilight" effect on recent photo, Save on SD card	1	2
Snapchat	Take snap	Log in, Take snap, Add caption, Set retention, Send snap to self, View it	–	1
	Take video	Log in, Take video, Pick color, Draw Line, Save to gallery, Add to story	–	–
	Find friend	Log in, Add friend from contacts, Allow Access	–	–
	Edit friend	Log in, Search friend "abc", Block "abc", Unblock "abc", Delete "abc"	–	–
		Total confirmations required (out of 18 use cases)	2	8

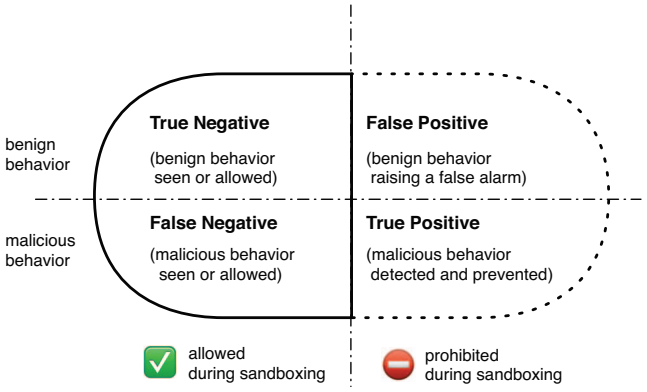

Figure 6: Confusion matrix. Program behavior is either benign or malicious; if it is not seen during mining (test generation), it is prohibited during sandboxing. The three risks are *false positives* (benign behavior not seen during testing and thus requiring confirmation during sandboxing), *false negatives allowed* (malicious behavior allowed because of too coarse sandbox rules), and *false negatives seen* (malicious behavior seen during mining, but not recognized as such, and thus allowed).

5. FINE-GRAINED ACCESS CONTROL

5.1 The Risks of Misclassification

User confirmations, as evaluated in Section 4.2, is only the first of the key questions we have to assess. In all generality, BOXMATE is an automatic system that decides on whether behavior should be allowed or not. As we do not assume a specification of what makes benign or malicious behavior, user confirmations, or false positives, is just one of two essential risks, illustrated in Figure 6:

False positive. A *false positive* occurs when benign behavior is mistakenly prohibited by the sandbox, degrading user experience and functionality. In our setting, a false alarm comes to be if some benign behavior is not seen during mining, and thus not added as allowed to the sandbox rules; it induces the access to be confirmed by the user. The number of confirmations can be reduced by *better testing* as well as *coarser-grained sandbox rules,* as evaluated in Section 4.2.

False negative. A *false negative* occurs when malicious behavior is mistakenly allowed by the sandbox, thus increasing the attack surface. In our setting, a false negative can come to be in two ways:

False negative allowed. The inferred sandbox rule may be *too coarse*, and thus allow future malicious behavior. This issue can be addressed by having *finer-grained sandbox rules,* as evaluated in Section 5.4.

False negative seen. The application may be malicious in the first place. Then, we risk to mine this malicious behavior during testing, such that it would get included in the sandbox rules. This issue can be addressed by *identifying malicious behavior during testing* already—a task made considerably simpler through the "disclose or die" principle imposed by BOXMATE (Section 6).

As with any classifier, a measure that decreases the rate of false negatives typically leads to a greater rate of false positives, and vice versa. Generally, the more benign behavior we see during mining and allow in our rules (true negatives), the fewer false alarms we will encounter during sandboxing. However, if the mined rules *overapproximate* and thus also allow possible malicious behavior, we increase the risk of false negatives. If the mined rules are too *specific,* though (say, only allow the *exact* behavior seen during mining), we again encounter false positives during sandboxing. In this section, we thus evaluate more fine-grained rules, with the aim of reducing the risk of a *false negative allowed* (Figure 6).

5.2 User-Driven Access Control

By default, BOXMATE simply checks whether the app as a whole uses the same APIs as found and distinguished during recording; we call this *per-app access control*. This policy allows for quick saturation during mining, and thus few false alarms during enforcement; however, it may be too coarse to prevent some attacks. For instance, once we have seen that SNAPCHAT can read contact phone numbers, *any function* within SNAPCHAT, including background tasks, would be allowed to do that. However, as we have seen in Figure 3, SNAPCHAT accesses phone numbers only to allow the user to find other SNAPCHAT users among his friends. How about restricting contact access to this functionality only?

To this end, BOXMATE implements a more *fine-grained* access control policy. During sandboxing, *per-event access control* also verifies whether the API call was triggered by *the same event* as during mining:

1. During mining, BOXMATE records pairs $(event, api)$, where *event* is the *identifier of the event-triggering GUI element* and *api* is the sensitive API called by the event handler.
2. During sandboxing, upon each call to a sensitive API api' triggered by a GUI element $event'$, BOXMATE checks whether $(event', api')$ was already found during mining; if not, the call is flagged.

Since our "events" are interactions with named GUI elements, and as our API calls all refer to user-owned resources, the BOXMATE per-event access control realizes the principle of *User-Driven Access Control* [33, 36], namely tying access to *user-owned resources* to *user actions* in the context of an application.

5.3 Distinguishing Events

Since we want to recognize earlier events, we need a means to uniquely identify an event. To this end, BOXMATE applies the following rules to identify events. All views (GUI elements) in Android have three features:

- A *resource identifier* r that associates views and programmatic actions ("`login_button`");
- A *text label* l possibly displayed on the screen ("Login");
- A *content description* d that can be read out loud to the user as an accessibility feature ("Login").

While most of these features are defined in an XML layout file, all of them can also be defined or changed at run time; hence the need for a dynamic analysis.

BOXMATE stores an event e as a tuple $e = (id, action)$:

id by default is the resource identifier r; if r is empty, $id = d$ instead; and if d is empty as well, $id = l$ instead. We prefer identifiers to labels since the latter may change during operation—for instance when changing the app's language.

action is the *user interaction* that triggers the event; for buttons, this is either a click or a long click.

With these rules, two buttons are different even if they sport the same text ("Ok"), as long as they have different resource identifiers.

The following rules apply for special events:

- If all of r, l, and d are empty, e has the special value *unlabeled*. All *unlabeled* events are treated as one.
- If the thread ID is not equal to 1 (the GUI thread), e has the special value *background*. Again, all *background* events are treated as one.
- If the app is reset (restarted), e has the special value *reset*. This captures events occurring during the program start.

5.4 Evaluation

Let us see how per-event access control works in our SNAPCHAT example. Within SNAPCHAT, the *contacts_permission_button* (API 9 in Figure 3) is the only trigger we found for accessing contacts or their phone numbers. Hence, enforcing per-event access control would always require that the user press this specific button before contacts can be accessed. Finding friends on SNAPCHAT is probably a rare, if not one-time only event for most users. Thus, even if an attacker worked around the restriction by manipulating this very functionality (say, by sending contact data to a different address), the attack surface is still greatly reduced.

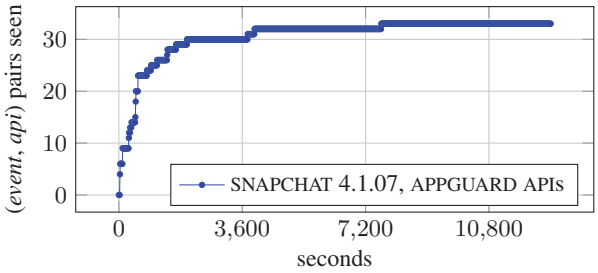

Figure 7: DROIDMATE per-event API saturation. After 60 minutes (3,600 seconds), DROIDMATE has discovered 32 unique (*event*, *api*)-pairs used by SNAPCHAT.

The downside of per-event access control is that it may raise false alarms more easily. This either translates into a *longer mining phase,* allowing DROIDMATE to find more $(event, api)$ pairs, or into a greater risk of false alarms. This is illustrated in Figure 7, showing the saturation of $(event, api)$ pairs during mining. In contrast to Figure 2, we see that it takes more than an hour of testing until the chart flattens at over 90% of all $(event, api)$-pairs ever explored. A similar late saturation can also be seen when mining $(event, api)$-pairs for the other twelve apps, as summarized in Figure 8.

Fine-grained policies take longer to mine.

How does the finer granularity impact false alarms? In Table 2, the "event" column shows the set of alarms encountered. We see that the higher granularity comes at the expense of six more confirmations: In the *barcoo* "Search for Product" use case, an unlabeled button not triggered during testing requests the current location. *CleanMaster* requires three confirmations: two for changes to configuration when a scan is started or a report is sent, and one when a handle to *PowerManager$WakeLock* is acquired after the scan is finished. *PicsArt* registers a content observer of *content://com.picsart.studio.provider/user.update* when a "gallery" button is pressed. In the SNAPCHAT "Take video" use case, a "status" button accesses the external media the video is saved in. In all cases, the alarm would be raised right after the user presses the appropriate button; the user thus is in the appropriate context to understand why the respective function requires access to, say, the location or the external media, and thus make an informed decision. This is similar to the model imposed upon users in Android 6—except that with BOXMATE, we can already eliminate most alarms during testing. Also, we require the permission anew for each button, not only once per app, thus further reducing attack surface.

Fine-grained policies increase the risk of false alarms.

On top, our model brings several additional benefits. By tying API calls to user interaction, any stealthy call from the background would be automatically prohibited. Thus, none of the apps could suddenly start sending text messages, turn the microphone on, track the location, or access sensitive contact or calendar data without the user initiating or acknowledging access.[5] We find this a nice thing to have, and answer **Q2:**

Fine-grained policies reduce the attack surface.

[5]The exception is if this is already part of the app's normal operation—as in *CleanMaster* and *VirusScan.*

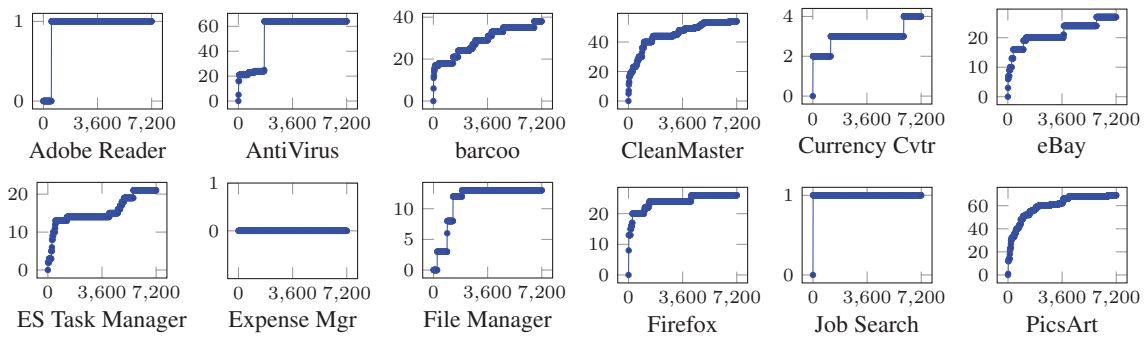

Figure 8: Per-event saturation for the apps in Table 1. As in Figure 7, the y axis is (*event*, *api*) pairs; the x axis is seconds spent.

6. ASSESSING SANDBOXES

At this point, we have one risk left—namely the risk of a *false negative seen* (Figure 6): If malicious behavior is already present during mining, the mined sandbox will not prevent it in the future. This feature is actually a strength of BOXMATE, as it puts malware writers into a "disclose or die" dilemma: Either the malware writer activates the malicious behavior during testing already—and only then will it be allowed during production; or she does not activate the behavior—and then the sandbox will prohibit it in the future. In practice, this means that even an attempt for malicious behavior *always will be detectable in the first place,* as the appropriate API calls will have to be made during testing and mining already, and eventually show up as sandbox rules.

While mined sandbox rules by themselves do not (and cannot) tell whether behavior is malicious or benign, or intended or unintended, they do *explicitly* record what an application does and what not as it comes to privacy. Mined sandboxes can thus assist in well-established techniques to assess behavior and establish trust:

Checking behavior. Anyone can mine a sandbox from a given app, checking which APIs are being used by which functionality; this alone already gives a useful overview about what the app does and why. As these rules come from concrete executions, one could easily assess concrete resource identifiers, such as file or host names, or URLs accessed. A mined sandbox easily serves as input for manual and automatic threat assessment.

Comparing and certifying sandboxes. As users and experts alike can mine sandboxes, they can also publish and compare their findings. This allows for independent certification and revalidation schemes, as well as trust networks. Again, anything not found will automatically be prohibited by the sandbox.

Open privacy. With the "disclose or die" dilemma, vendors would also be motivated to disclose app behavior as it comes to resources being accessed. In the long run, this would lead to open discussions of what all apps do in terms of privacy; very much as in the open source movement, but without forcing vendors to disclose their source code.

Mining normal behavior. We have designed BOXMATE to be easily applicable to arbitrary binaries. We can thus automatically assess *large sets of apps,* extracting *rules of normal behavior* that may even be tied to app descriptions [22].

These features can all be helpful in answering the third and last key question, namely whether mined sandboxes can help to assess behavior—and thus prevent the risk of a *false negatives seen* (Figure 6). Since at this point, the ability of sandboxes to assess and compare behavior is only secondary, a full-fledged evaluation is beyond the scope of this paper. However, let us give an example of how sandbox mining helps to assess program behavior.

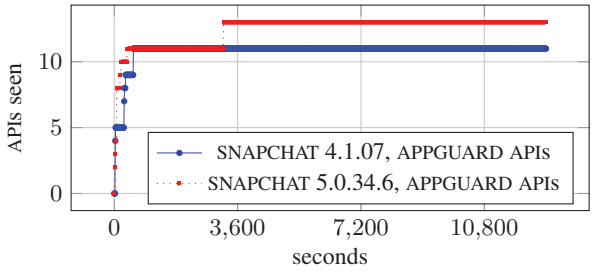

Figure 9: DROIDMATE per-app API saturation, comparing SNAPCHAT versions. The upper thin red line is the newer SNAPCHAT 5 version.

SNAPCHAT version 5, released in February 2015, is a redesign of the original SNAPCHAT version 4 described in this paper. We have run BOXMATE on the new SNAPCHAT version, comparing the resulting sandbox with the original sandbox as mined for version 4.

Figure 9 contrasts the API saturation charts for the two versions; we can see that the new SNAPCHAT 5 accesses the same amount of sensitive resources as the old SNAPCHAT 4 version, but the APIs are somewhat different. Overall, we found SNAPCHAT 5 accesses four sensitive APIs not seen during the exploration of SNAPCHAT 4:

1. Usage of the Android 4.x *AudioRecord* interface (the old version used the Android 1.x *MediaRecorder* interface instead)
2. Read/Write access to image thumbnails through methods of *ContentResolver* interface like *query()*, *openFileDescriptor()* and *insert()*, while reading the SNAPCHAT privacy policy.
3. Access of the user's *line1* phone number (after clicking on the *mobile_number* button)
4. Usage of the Android 4.x *PowerManager* interface, forcing the device screen to stay on while the message is sent (*send_to_bottom_panel_send_button* button).

Since DROIDMATE records the events associated with each call, we can place each API into context, and thus determine why they would be required. The most sensitive data, the user's phone number, is only accessed after the user has clicked on the appropriate button, acknowledging access. Just as we compared the respective sandboxes to determine what has changed in SNAPCHAT, any expert could also have determined other changes between old and new versions of possibly less trustable programs; the differences could even be presented in a form amenable for end users.

A user wishing to preserve privacy settings could also run the untrusted SNAPCHAT 5 version within the trusted sandbox mined

from SNAPCHAT 4, with any new API accesses being detected by the SNAPCHAT 4 sandbox. Then, she would have to confirm access once in each of the four situations:

1. When recording audio (for the *AudioRecord* interface),
2. When sending a message (for the *PowerManager* interface),
3. When reading the SNAPCHAT privacy policy, and
4. When sending the message, forcing the device to stay on.

Each case would inform the user that there is a new feature—and thus enable her to detect, assess, and prevent potentially malicious behavior changes.[6] Our answer to **Q3** is thus positive:

Mined sandboxes can help in assessing and comparing app behavior.

7. THREATS AND LIMITATIONS

Although our results demonstrate the principal feasibility of sandbox mining, we would not generalize our findings into external validity. Our sample of programs is small, is all on Android, and all GUI-based. For other programs and platforms, we may have to devise different or additional test generators, possibly requiring models of the program input structure as well as the sensitive resources to be monitored and protected. These test generators may be less successful in exploring program behavior, leading to more false alarms.

The set of use cases we have compiled for assessing the risk of false alarms (Table 2) does not and cannot cover the entire range of functionality of the analyzed apps. While we assume that the listed use cases represent the most important functionality, other usage profiles may yield different results.

Finally, keep in mind that in the absence of a specification, a mined policy can never express whether behavior is benign or malicious; and thus, our approach cannot eliminate the risks of both false alarms and missed attacks. However, by detecting and preventing unexpected changes, our approach is set to reduce both these risks, even in the absence of specifications. On top, existing specifications for benign or malicious behavior would be very easily integrated.

8. CONCLUSION AND FUTURE WORK

The purpose of testing always has been to detect *abnormal* behavior. In this work, we give testing a new purpose, namely to extract *normal* behavior—a task that testing arguably is much better suited to, and even more so in the security domain. By excluding *behavior not seen during testing,* we turn the incompleteness of testing into a guarantee that bad things not seen so far cannot happen. This principle of *test complement exclusion* works well in practice: In our experiments, automatic test generators sufficiently covered program behavior, reducing the risk of false alarms. Furthermore, fine-grained per-event access rules can be used to further reduce the attack surface, and the mined sandbox rules can help to assess program behavior, both reducing the risk of false negatives. All in all, we thus obtain a *fully automatic solution to security promising several benefits at little cost.*

Besides general goals such as robustness and scalability, our future work will focus on the following topics:

Better test generation. Any improvement in automatic test generation—where "improvement" is not so much the ability to detect bugs, but rather coverage of "normal" behavior—will decrease the number of false alarms. The long-term goal is to explore behavior as quickly as a human tester would.

Alternate input sources. Test generation for Android apps is made easy by the fact that the GUI and its structure are easily accessible and explorable for test generators. We are investigating novel ways of inferring input structure from arbitrary programs and input sources, such that these input sources can be triggered, too.

Access control policies. Mining tighter and more detailed security policies will catch more unexpected "abnormal" behavior. We are exploring further policies, involving file or network names accessed, callers, call sequences, GUI sequences, or information and data flow from and to sensitive resources; and in all cases, we have to search for sweet spots that minimize both the attack surface and the number of false alarms.

Remining at runtime. If some application functionality is available only after specific interaction (e.g., a login/password combination, an in-app purchase, or a special code to enter a maintenance mode), we might not see it during mining. One possible way to overcome this issue could be to re-mine new functionality once user executes the interaction.

Computed resources. On platforms like UNIX and WINDOWS, sensitive resources are accessed as *files* whose paths would be computed at runtime from configuration files, environment variables, and other external influences. We are working on sandbox rules that express variability across configurations, yet are tight enough to keep the attack surface small.

Threat models. The evaluation of all these options will require systematic and objective evaluation. Besides further expanding our use cases, we are working on creating benchmarks for typical threats, such that we can automatically assess the effectiveness of the above options.

For more information on DROIDMATE and BOXMATE, including source code as well as all experimental data, see our site

`http://www.boxmate.org/`

Acknowledgments. Michael Backes, Marcel Böhme, Juan Pablo Galeotti, Alessandra Gorla, and Christian Rossow provided useful feedback on earlier revisions of this paper. Ahmad Shahzad assisted in feasibility studies. Florian Gross and Konstantin Kuznetsov provided app binaries. This work was funded by an European Research Council (ERC) Advanced Grant "SPECMATE – Specification Mining and Testing".

9. REFERENCES

[1] Android 6 permission system. `https://developer.android.com/preview/features/runtime-permissions.html`. Retrieved 2015-08-27.

[2] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. 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* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 259–269.

[3] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., GILL, P., AND LIE, D. Short paper: A look at smartphone permission models. In *Proceedings of the 1st ACM Workshop on*

[6] Note that whether the user sees these alarms as "false" entirely depends on the trust the user puts in the new SNAPCHAT version.

Security and Privacy in Smartphones and Mobile Devices (New York, NY, USA, 2011), SPSM '11, ACM, pp. 63–68.

[4] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. PScout: Analyzing the Android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 217–228.

[5] BACKES, M., BUGIEL, S., HAMMER, C., SCHRANZ, O., AND VON STYP-REKOWSKY, P. Boxify: Full-fledged app sandboxing for stock android. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* (2015), pp. 691–706.

[6] BACKES, M., GERLING, S., HAMMER, C., MAFFEI, M., AND VON STYP-REKOWSKY, P. AppGuard–fine-grained policy enforcement for untrusted Android applications. In *Data Privacy Management and Autonomous Spontaneous Security*, J. Garcia-Alfaro, G. Lioudakis, N. Cuppens-Boulahia, S. Foley, and W. M. Fitzgerald, Eds., Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 213–231.

[7] BALIGA, A., GANAPATHY, V., AND IFTODE, L. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 2008 Annual Computer Security Applications Conference* (Washington, DC, USA, 2008), ACSAC '08, IEEE Computer Society, pp. 77–86.

[8] BARTEL, A., KLEIN, J., LE TRAON, Y., AND MONPERRUS, M. Automatically securing permission-based software by reducing the attack surface: An application to Android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2012), ASE 2012, ACM, pp. 274–277.

[9] BHORASKAR, R., HAN, S., JEON, J., AZIM, T., CHEN, S., JUNG, J., NATH, S., WANG, R., AND WETHERALL, D. Brahmastra: Driving apps to test the security of third-party components. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.* (2014), pp. 1021–1036.

[10] BIERMA, M., GUSTAFSON, E., ERICKSON, J., FRITZ, D., AND CHOE, Y. R. Andlantis: Large-scale Android dynamic analysis. *CoRR abs/1410.7751* (2014).

[11] BLÄSING, T., BATYUK, L., SCHMIDT, A.-D., CAMTEPE, S., AND ALBAYRAK, S. An Android application sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on* (Oct 2010), pp. 55–62.

[12] BURGUERA, I., ZURUTUZA, U., AND NADJM-TEHRANI, S. Crowdroid: Behavior-based malware detection system for Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 15–26.

[13] CHANDOLA, V., BANERJEE, A., AND KUMAR, V. Anomaly detection: A survey. *ACM Comput. Surv. 41*, 3 (July 2009), 15:1–15:58.

[14] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.

[15] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev. 35*, 5 (Oct. 2001), 57–72.

[16] ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering* (New York, NY, USA, 1999), ICSE '99, ACM, pp. 213–224.

[17] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 627–638.

[18] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security* (New York, NY, USA, 2012), SOUPS '12, ACM, pp. 3:1–3:14.

[19] FORREST, S., HOFMEYR, S. A., SOMAYAJI, A., AND LONGSTAFF, T. A. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1996), SP '96, IEEE Computer Society, pp. 120–.

[20] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In *Proceedings of Network and Distributed Systems Security (NDSS 2008)* (July 2008), pp. 151–166.

[21] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Sage: Whitebox fuzzing for security testing. *Queue 10*, 1 (Jan. 2012), 20:20–20:27.

[22] GORLA, A., TAVECCHIA, I., GROSS, F., AND ZELLER, A. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, ACM, pp. 1025–1035.

[23] HAO, S., LIU, B., NATH, S., HALFOND, W. G., AND GOVINDAN, R. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2014), MobiSys '14, ACM, pp. 204–217.

[24] HU, C., AND NEAMTIU, I. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test* (New York, NY, USA, 2011), AST '11, ACM, pp. 77–83.

[25] LINDORFER, M., NEUGSCHWANDTNER, M., WEICHSELBAUM, L., FRATANTONIO, Y., VAN DER VEEN, V., AND PLATZER, C. ANDRUBIS – 1,000,000 apps later: A view on current Android malware behaviors. In *Proc. 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)* (2014), ACM.

[26] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 229–240.

[27] MACHIRY, A., TAHILIANI, R., AND NAIK, M. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2013), ESEC/FSE 2013, ACM, pp. 224–234.

[28] MAHMOOD, R., ESFAHANI, N., KACEM, T., MIRZAEI, N., MALEK, S., AND STAVROU, A. A whitebox approach for automated security testing of Android applications on the cloud. In *Proceedings of the 7th International Workshop on Automation of Software Test* (Piscataway, NJ, USA, 2012), AST '12, IEEE Press, pp. 22–28.

[29] MILLER, B. P., FREDRIKSEN, L., AND SO, B. An empirical study of the reliability of UNIX utilities. *Commun. ACM 33*, 12 (Dec. 1990), 32–44.

[30] Monkey: UI/Application exerciser. `http://developer.android.com/tools/help/monkey.html`. Retrieved 2015-02-01.

[31] NEUNER, S., VAN DER VEEN, V., LINDORFER, M., HUBER, M., MERZDOVNIK, G., MULAZZANI, M., AND WEIPPL, E. R. Enter sandbox: Android sandbox comparison. *CoRR abs/1410.7749* (2014).

[32] PROVOS, N. Improving host security with system call policies. In *Proc. USENIX Security* (2003), USENIX Association, pp. 18–32.

[33] ROESNER, F., KOHNO, T., MOSHCHUK, A., PARNO, B., WANG, H. J., AND COWAN, C. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 224–238.

[34] SALTZER, J., AND SCHROEDER, M. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (Sept 1975), 1278–1308.

[35] SHABTAI, A., KANONOV, U., ELOVICI, Y., GLEZER, C., AND WEISS, Y. "Andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems 38*, 1 (2012), 161–190.

[36] SHIRLEY, J., AND EVANS, D. The user is not the enemy: Fighting malware by tracking user intentions. In *Proceedings of the 2008 Workshop on New Security Paradigms* (New York, NY, USA, 2008), NSPW '08, ACM, pp. 33–45.

[37] SOMMER, R., AND PAXSON, V. Outside the closed world: On using machine learning for network intrusion detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 305–316.

[38] ZELLER, A. Test complement exclusion: Guarantees from dynamic analysis. In *Proc. International Conference on Program Comprehension (ICPC)* (2015). Abstract of invited keynote.