The Journal of Systems and Software 210 (2024) 111949

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Automatic testing of runtime enforcers with Test4Enforcers[☆]

Oliviero Riganelli [*], Daniela Micucci, Leonardo Mariani

University of Milano - Bicocca, Milan, Italy

ARTICLE INFO

Keywords:
Runtime enforcement
Testing enforcers
Test case generation
Android apps

ABSTRACT

Users regularly use apps to access services in a range of domains, such as health, productivity, entertainment, and business. The safety and correctness of the runtime behavior of these apps is thus a key concern for users. Indeed, unreliable apps may generate dissatisfaction, frustration and issues to users.

Runtime enforcement techniques can be used to implement *software enforcers* that monitor executions and apply corrective actions when needed, potentially preventing misbehaviors and failures. However, enforcers might be faulty themselves, applying the wrong actions or missing to apply the right actions.

To address this problem, this paper presents *Test4Enforcers*, an approach to automatically test software enforces. Test4Enforcers relies on an enforcement model describing the strategy that shall be applied at runtime to correct misbehaviors. Test4Enforcers first uses the enforcement model to derive a specification of the test cases that shall be executed to validate any software enforcer implemented from the given model. Then, it automatically turns the test specification into a set of concrete test cases that can be executed against apps augmented with the enforcers.

We evaluated Test4Enforces with a set of 3,135 faults injected in the enforcers derived from 13 enforcement models. Results show that Test4Enforcers can automatically reveal 64% of the faults, while existing approaches relying on crash detection can only reveal 6% of the faults. Test4Enforcers is also practical since testing an enforcer required 9 min, in the worst case.

1. Introduction

Context. Mobile apps are extremely popular, with a huge number of apps available through well-established marketplaces.[1] Interestingly, in 2021 the number of mobile users worldwide was 7.1 billion, and forecasts indicate that this number is set to rise to 7.26 billion by the end of 2022 (Taylor, 2023). The number of mobile app downloads worldwide has been steadily increasing, rising from 140.7 billion in 2016, exceeding 200 billion in 2019, and reaching 230 billion in 2021 (Ceci, 2023). This has formed a large new market for mobile apps, which is expected to generate more than $613 billion dollars in revenue in 2025 (Ceci, 2023). In fact, in 2019 alone, the major mobile app stores took in a total of 4.5 million new apps and people spent $120 billion in app stores worldwide alone (Koetsier, 2020).

Motivation. Ensuring safe runtime behavior is extremely important for the reliability and trust of mobile apps. An unreliable app may generate unexpected crashes causing frustration to the users, while an unsafe app may leak important personal data to third parties who have not the right to access it. In order to make apps more reliable, Runtime Enforcement (RE) techniques can be used to monitoring the behavior of an application at runtime and take corrective actions, if any correctness property is violated (Falcone et al., 2011). For instance, RE can detect an erroneous interaction with an API and change it on-the-fly to prevent a crash (Riganelli et al., 2019), or can stop an undesired data leak before the data is leaked (Rasthofer et al., 2014). Indeed, RE has been applied to address many different types of problems, such as security (Falcone et al., 2012), resource management (Riganelli et al., 2019), data leak (Rasthofer et al., 2014), and safety (Könighofer et al., 2022).

RE techniques typically consist of three key components: a *policy*, which describes a property that must not be violated during execution (e.g., a specification of the protocol that must be satisfied to correctly interact with an API); an *enforcement model*, which rigorously represents the enforcement strategy that is applied at runtime to modify an execution that may violate the policy into a correct one (e.g., to specify that a call to a deprecated method must be systematically replaced with the invocation of a different method); and a *software enforcer*, which implements the strategy represented in the enforcement model

☆ Editor: Yan Cai.
* Corresponding author.
E-mail address: oliviero.riganelli@unimib.it (O. Riganelli).
1 https://play.google.com/store, https://www.apple.com/app-store

https://doi.org/10.1016/j.jss.2023.111949
Received 8 March 2023; Received in revised form 29 November 2023; Accepted 27 December 2023
Available online 2 January 2024

in a software component that operates it (e.g., to actually intercept and replace method calls).

It is particularly important that a software enforcer operates correctly. An incorrect software enforcer may inadvertently turn policy violations into even more severe failures and crashes. Indeed, there are several possible sources of problems for an enforcer. First, the *enforcement model might be inaccurate*. Although the model can be verified before the corresponding software is (semi)-automatically generated, the model, and thus the verification activity (Riganelli et al., 2017b), abstracts from many potentially relevant details that might lead to software enforcers that do not always behave correctly. For instance, a model that abstracts the values used to interact with the methods implemented by a given API may lead to the development of a software enforcer that does not manage parameters correctly. Second, the *implementation might be inconsistent* with the model, either because the code generation process is not automatic and the manually implemented software might not be conformant with the model, or because the code added to an automatically generated implementation may introduce undesired side-effects.

Objective. Establishing the correctness of a software enforcer is a big challenge, demanding validation across diverse scenarios, including cases where the enforcer is applied to both compliant and non-compliant apps. This paper addresses this challenge by introducing *Test4Enforcers*, the first automated approach for rigorously testing software enforcers. Test4Enforcers is an approach to automatically test software enforcers. An open-source tool called Test4Enforcers has been developed to implement the Test4Enforcers approach. Test4Enforcers seamlessly generates test specifications from enforcement models, subsequently transforming them into concrete test cases. These test cases are then executed on mobile apps, with and without the corresponding enforcer derived from the same enforcement model. This enables a comparative analysis of their execution in order to identify faults or incorrect behavior in the enforcer.

Research method and questions. To investigate the effectiveness of Test4Enforcers, we adopted a *quantitative research methodology*. Indeed, we expect a test case generation approach for software enforcers to be able to deliver clear quantitative evidence of its effectiveness and efficiency in revealing faults. To deliver this evidence, we investigated the following research questions:

- **RQ1 (Effectiveness):** *Can Test4Enforcers reliably detect faults in software enforcers?* We assess the capability of Test4Enforcers to pinpoint faults in software enforcers by seeding faults into enforcers derived from enforcement models designed for well-established Android policies.
- **RQ2 (Efficiency):** *How efficiently can Test4Enforcers validate software enforcers?* This research question evaluates Test4Enforcers' efficiency by measuring the time required to test the enforcers studied in RQ1.

The research method involves the following steps:

1. **Test Specification Generation:** Test4Enforcers generates abstract test specifications from enforcement models. These specifications describe the expected behavior of the enforcer under various conditions.
2. **Concrete Test Case Generation:** The abstract test specifications are transformed into concrete test cases. These test cases represent specific scenarios to be executed on mobile apps.
3. **Fault Injection:** To assess Test4Enforcers' effectiveness, we intentionally inject faults into the enforcer, simulating potential inaccuracies and inconsistencies. This is achieved through mutation testing, where artificial faults (mutations) are introduced into the enforcers. The goal is to verify Test4Enforcers' capability to reliably detect these mutations.
4. **Testing Phase:** The generated concrete test cases are executed on both the mobile apps and the apps augmented with faulty enforcers. Test4Enforcers captures and compares the behavior of the two scenarios in order to detect faults in the enforcer.
5. **Data Analysis:** The results are analyzed to determine the percentage of faults identified by Test4Enforcers and to assess the efficiency of the testing process.

The data collected for RQ1 includes the percentage of faults accurately identified by Test4Enforcers. This data provides insights into the reliability and effectiveness of the testing approach. For RQ2, the data collected involves the time required to test the enforcers, allowing us to draw conclusions about the efficiency and practicality of Test4Enforcers in real-world scenarios.

Test4Enforcers successfully identified 64% of 3,135 problematic enforcers with seeded faults across a variety of scenarios. The quantitative evidence demonstrates the effectiveness of Test4Enforcers in pinpointing faults in software enforcers. The efficiency evaluation, measuring testing times, indicates practical efficiency, ranging from approximately 2 to less than 10 min for different enforcers. These results demonstrate the capability of Test4Enforcers in efficiently and effectively detecting and exposing enforcer faults.

Contribution. This paper extends our former paper on testing of software enforcers (Guzman et al., 2020) by introducing a more rigorous and detailed presentation of the approach, new examples, and a large experimental evaluation based on 3,135 faulty enforcers.

The main contributions of this paper are:

- The definition of Test4Enforcers, a test case generation approach for software enforcers that implement enforcement strategies encoded with enforcement models. Test4Enforcers includes both the capability to generate (abstract) test specifications from enforcement models and concrete test cases from test specifications.
- The experimental evaluation of the approach with 3,135 faulty injected into 13 enforcement models designed to enforce well-documented API correctness policies. The faulty enforcers have been used to augment nine apps that use the APIs relevant to the enforcers, demonstrating the effectiveness and efficiency of the approach.

Additionally, it is worth noting that Test4Enforcers has been released as an open-source tool, and the experimental material required to replicate the results reported in this paper are available publicly at https://github.com/lta-unimib/Test4Enforcers.

This work is beneficial to both practitioners and researchers in the field of mobile computing, runtime enforcement and software testing. Practitioners can exploit our enforcement testing approach, Test4Enforcers, to assess and enhance the correctness of software enforcers within their mobile platforms. By doing so, they can improve the quality and reliability of their systems, and the satisfaction of the users. Researchers, on the other hand, can leverage our results to advance their studies in software engineering, testing, and runtime enforcement. In particular, our work provides a foundation for further research in software enforcement testing and offers a benchmark for evaluating future researches in this area. Indeed our results, albeit encouraging, show the need of designing better enforcement testing approaches to effectively reveal faulty enforcement strategies. In summary, this work provides practical benefits for developers and opens up new research opportunities for researchers in the field.

Structure. The paper is organized as follows. Section 2 introduces a running example that is used throughout the paper to illustrate the approach. Section 3 provides background information about runtime policy enforcement. Section 4 presents Test4Enforcers. Section 5 presents the design of the empirical evaluation, whose results are described in Section 6. Section 7 discusses related work. Finally, Section 8 provides our conclusions and research directions.

2. Running example

In this section, we present a running example based on the *Plumeria* app, which is an Android app for capturing pictures that we downloaded from git (https://github.com/DonLiangGit/Plumeria), and the policy that specifies how to correctly use the Android Camera Library to take pictures (Android Docs, 2023a). We selected this app and policy because they are simple, but yet complex enough to illustrate the key concepts related to testing software enforcers.

Listing 1 shows a code-snippet of `MyActivity`, which is the Android activity[2] that uses the `Camera` in Plumeria. On startup, the callback method `onCreate()`[3] is executed and a new `Camera` object is created by method `getCamera()`, whose implementation acquires the backward camera of the device invoking the method `Camera.open()`. The library method `Camera.open()` is used to access the camera, which can be later released by invoking the library method `Camera.release()`.

The public list of issues for this app reports an incorrect use of the camera.[4] In fact, every time `MyActivity` becomes invisible, the camera becomes inaccessible to every app in the device. The camera is not even anymore accessible from Plumeria because, when `MyActivity` is visible again, the `onCreate()` callback method is executed, and the call to `Camera.open()` produces an exception, since the app is attempting to acquire a camera that is already in use (by the app itself).

Listing 1: A code-snippet that shows the activity that handles the camera in Plumeria

```
...
public class MyActivity extends TabActivity {

    private Camera camInstance;
    ...

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_my);

        camInstance = getCamera();
        cameraPreview = new CameraPreview(this, camInstance);
        preview = (FrameLayout) findViewById(R.id.camera_preview);
        preview.addView(cameraPreview);
        ...
    }

    public static Camera getCamera() {
        Camera cam = null;
        cam = Camera.open();
        return cam;
    }

    ...
    @Override
    public void onPause() {
        super.onPause();
    }
    ...
}
```

This fault is caused by the app that does not handle the access to the camera properly. In particular, the `onPause()`[5] callback method, which is invoked when the activity becomes invisible, does not invoke the method `Camera.release()`. Indeed, the Android API documentation provides a policy for camera release that specifies that anytime an app stops using the camera, the app must explicitly release the camera to make it available to the other apps (Android Docs, 2023b). More precisely, if an activity is using the camera and the activity receives an invocation to the callback method `onPause()`, the activity must release the camera.

In the rest of the paper, we exploit this faulty app as a running example to present the concept of software enforcers and to show how Test4Enforcers can automatically test a software enforcer implemented to enforce the camera release policy.

3. Background

In this section we introduce the notion of runtime policy, policy enforcement, and software enforcer.

3.1. Runtime policy

We define software systems at a higher level of abstraction, as also done in Ligatti et al. (2005). A system S is specified by a set of observable program actions Σ. An *execution* σ is a finite sequence of actions $a_1; a_2; \dots; a_n$. Σ^* is the set of all finite sequences of actions on a system with action set Σ. Given a set of executions $\chi \subseteq \Sigma^*$, a *policy* is a predicate P on χ. A policy P is satisfied by a set of executions χ if and only if $P(\chi)$ is true. The definition of a policy requires a predicate P defined over all finite executions. Following the distinction introduced in Ligatti et al. (2005), we distinguish between properties and policies as follows: A policy P is a property if and only if there exists a characteristic predicate $\hat{P}$ over Σ^* such that for all $\chi \in \Sigma^*$, the following is true: $P(\chi) \iff \forall\sigma \in \chi : \hat{P}(\sigma)$. Consequently, a property is exclusively defined with regard to individual executions, serving as a computable predicate applicable solely to finite sequences of executions. It does not entail a specification of relationships between different program executions. The distinction between properties and policies holds significant importance when reasoning about enforcers, as enforcers observe only individual executions and, thus, can enforce only properties, not more general policies. There exists a direct one-to-one correspondence between a property P and its characteristic predicate $\hat{P}$, so we use the notation $\hat{P}$ unambiguously to refer both to a characteristic predicate and the property it induces.

In addition, we introduce the formalism of a *Policy automaton* to provide a rigorous framework for evaluating runtime policies: A policy automaton A is a tuple $\langle \Sigma, S, s_0, \delta, F \rangle$ where Σ is a finite set of program actions, S is a finite set of states, $s_0 \in S$ is the initial state, $\delta : S \times \Sigma \to S$ is a transition function, and $F \subseteq S$ is a set of accepting states. An execution of A on a finite sequence of actions $\sigma = a_1; a_2; \dots; a_n$ is a sequence of states $s_1; s_2; \dots; s_n$ such that $s_{i+1} = \delta(s_i, a_i)$. A finite execution is accepting if the last state of the run is an accepting state. A property $\hat{P}$ is represented as a Policy automaton A if and only if $\forall\sigma \in \chi : \hat{P}(\sigma) \iff A$ accepts σ.

Policy of the running example. The Android framework includes the `Camera` API for capturing pictures and videos. To use the camera in their applications, developers must obtain an instance of a `Camera` by invoking the class method `open()`. The acquired camera instance can be released by invoking the instance method `release()`.

According to the Android documentation, to make the camera available to other applications and to avoid resource leaks, the usage of the `Camera` should be governed by the following policy:

Policy 1: "*Your application can make use of the camera after getting an instance of `Camera`, and you must be particularly careful to release the camera object when your application stops using it, and as soon as your application is paused (`Activity.onPause()`).*" Android Docs (2023b)

Fig. 1 shows a Policy automaton for the Policy 1. The automaton must accept all sequences that do not open the camera or all sequences that when the camera is opened it is also released before the `onPause()` method invocation. Otherwise, if after the `open()` method there is an `onPause()` method instead of a `release()` method, the policy is violated and the automaton halts. If there is any method invocation other than `onPause()` after the execution of `open()` method, the automaton simply waits for the `release()` method.

[2] Activities are fundamental components of Android apps and they represent the entry point for a user's interaction with the app https://developer.android.com/guide/components/activities.

[3] `onCreate()` is a callback method that is invoked by the Android framework when the activity is first created.

[4] See issue https://github.com/DonLiangGit/Plumeria/issues/1.

[5] `onPause()` is a callback method that is invoked by the Android framework when an activity is paused.

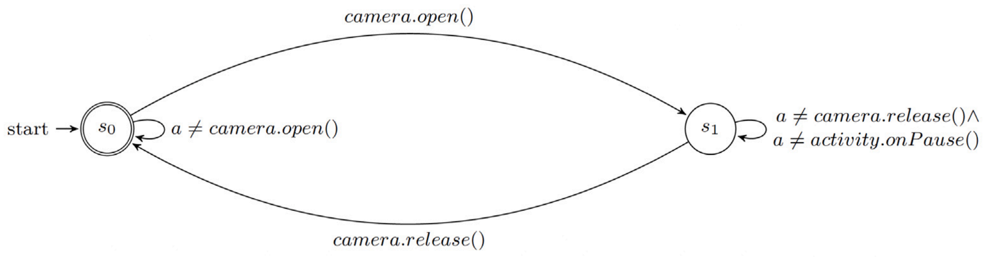

Fig. 1. Policy automaton of Policy 1.

3.2. Policy enforcement models

A policy enforcement model is a model that specifies how an execution can be altered to make it comply with a given policy. Policy enforcers can be represented with edit automata (Ligatti et al., 2005) that specify how a program execution is transformed by inserting and suppressing actions. Formally an *Edit Automaton* A is a finite-state machine $\langle \Sigma, S, s_0, \delta \rangle$, where:

- Σ is a finite set of actions,
- S is a finite set of states,
- s_0 is the initial state,
- and $\delta : S \times \Sigma \rightarrow S \times \Sigma^*$ is the transition function that maps a state and an action to the new state reached by the automaton, and the finite or empty sequence of actions emitted by the automaton. When the emitted action is the same as the accepted action, the automaton does not affect the execution. In the other cases, the actions that are actually executed are influenced by the edit automaton. Action suppression is represented with the empty sequence.

Let A be an edit automaton, a sequence $a_1, \ldots, a_k$ with $a_j \in \Sigma, \forall j = 1 \ldots k$ is an input sequence for state $s \in S$, if there exist states $s_1, \ldots, s_{k+1}$ such that $s = s_1$ and $\delta(s_i, a_i) = \langle s_{i+1}, o \rangle$, $\forall i = 1 \ldots k$, where $o \in \Sigma^*$ is a sequence of outputs emitted by the automaton when accepting a_i in state s_i. The input function $\Omega(s)$ is used to denote all input sequences of any length defined from a state $s \in S$. Similarly, given a state s and an input sequence $\gamma = \langle a_1, \ldots, a_k \rangle \in \Omega(s)$, the output function $\lambda(s, \gamma)$ denotes the output sequence $o_1, \ldots, o_m$ with $o_j \in \Sigma, \forall j = 1 \ldots m$ emitted by the automaton when accepting γ from a state $s \in S$.

In Fig. 2 we show the model of an enforcer specified as edit automaton that addresses the before-mentioned policy about the Camera. In a transition, the symbol before the slash indicates the input accepted by the automaton, while the sequence after the slash indicates the sequence of outputs emitted by the automaton when the input is accepted. The inputs are requests intercepted at runtime (these events are labeled with the *req* subscript) by the software enforcer and the outputs are the events emitted by the enforcer in response to the intercepted requests (these events are labeled with the *api* subscript). When the label of the output is the same than the label of the input (regardless of the subscript), the enforcer is just forwarding the requests without altering the execution. If the output is different from the input, the enforcer is manipulating the execution suppressing and/or adding requests. In the example in Fig. 2, when the current state is state s_0, the Camera has not been acquired yet and the `activity.onPause()` callback can be executed without restrictions. If the Camera is acquired by executing `camera.open()` (transition from s_0 to s_1), the camera must be released before the activity is paused (as done in the transition with the input `activity.release()` from s_1 to s_0). If the activity is paused before the camera is released, the enforcer modifies the execution emitting the sequence `camera.release()` `activity.onPause()`, which guarantees that the policy is satisfied despite the app is not respecting it.

Edit automaton is a natural option to describe software enforcers (Ligatti et al., 2009, 2005) since it is designed to specify how sequences of events can be modified at run-time, and it has also a rigorously defined syntax and semantics that can be exploited to define rigorous test case generation techniques that are guaranteed to target all the relevant combination of events represented in the model.

3.3. Implementation of the software enforcer

The enforcement strategy represented in the policy enforcement model must be translated into a suitable software component, the *software enforcer*, to achieve runtime enforcement. As any type of software, the software enforcer is not free of bugs. In particular, there are two classes of faults that may affect the enforcer:

- *Model Inaccuracies:* Although the designer may believe the enforcement model is correct, the strategy might end up being incorrect. For instance, the enforcer in Fig. 2 releases the camera when an activity is paused but does not acquire the camera back when the execution of the activity is resumed. This is a source of problems when the activity does not automatically re-acquire the camera once resumed. The strategy is thus inaccurate and should be extended to also include this stage. For simplicity in this paper we use the enforcer in Fig. 2 without complicating the model with the part necessary to acquire again a forcefully released camera.
- *Inconsistent Implementations:* The model must be translated into working code. Concerning the behavior specified in the model, the corresponding code can be easily produced automatically, thus preventing inconsistencies (unless the generator is faulty). If the code is implemented manually, its conformance with the model has to be verified. In both cases, additional code that deals with the aspects that are abstracted in the model must be implemented (e.g., to deal with the context and method parameters), and then validated.

Test4Enforcers automatically generates tests that target the above listed issues.

4. Test4Enforcers

Test4Enforcers represents a novel approach in the realm of software enforcer testing, designed to ensure the correctness and reliability of runtime enforcement (RE) in mobile applications. In this section, we provide a detailed description of the Test4Enforcers approach, detailing its key steps and outputs. At its core, Test4Enforcers automates test case generation, enabling the rigorous assessment of software enforcers across a diverse set of mobile apps, as shown in Fig. 3. Test4Enforcers employs a set of mobile applications as its testing scenarios. These apps serve as the testing grounds for evaluating the correctness and effectiveness of software enforcers, which are responsible for enforcing specific runtime policies. By leveraging real-world applications, Test4Enforcers assesses how enforcers behave in practice, considering various usage scenarios, app interactions, and potential deviations from expected behavior. This approach aims to enhance the trustworthiness and robustness of software enforcers, safeguarding both developers and end-users from unforeseen misbehavior and vulnerabilities.

$$\begin{tikzcd}[column sep=huge]
\text{start} \arrow[r] & s_0 \arrow[loop right, "activity.onPause()_{req}/activity.onPause()_{api}"] \arrow[rr, bend left=20, "camera.open()_{req}/camera.open()_{api}"] & & s_1 \arrow[ll, bend right=60, "activity.onPause()_{req}/camera.release()_{api};activity.onPause()_{api}"'] \arrow[ll, bend left=60, "camera.release()_{req}/camera.release()_{api}"]
\end{tikzcd}$$

Fig. 2. Enforcer for systematically releasing the camera when the activity is paused.

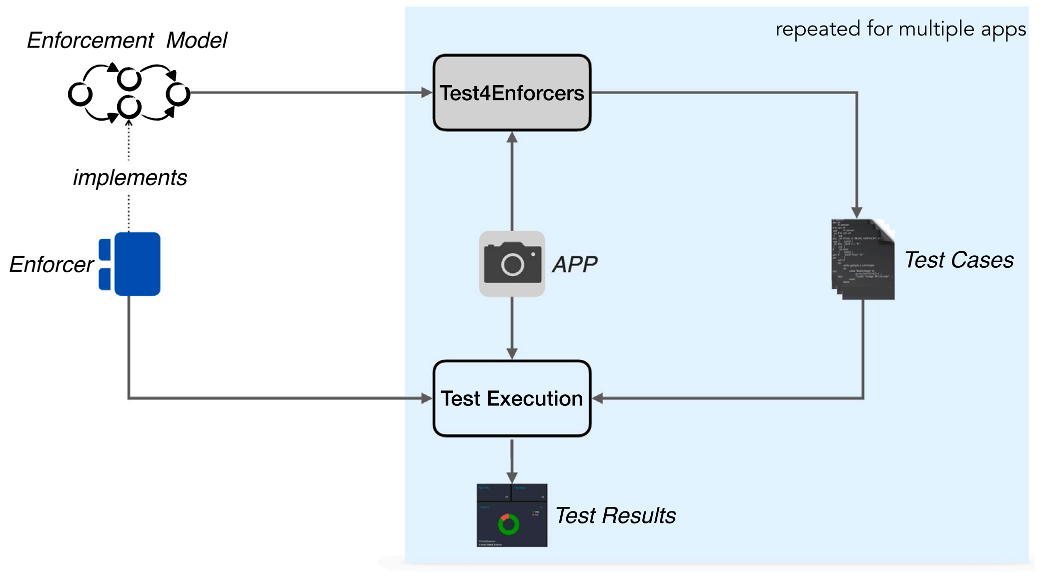

Fig. 3. Assessment of software enforcers with Test4Enforcers.

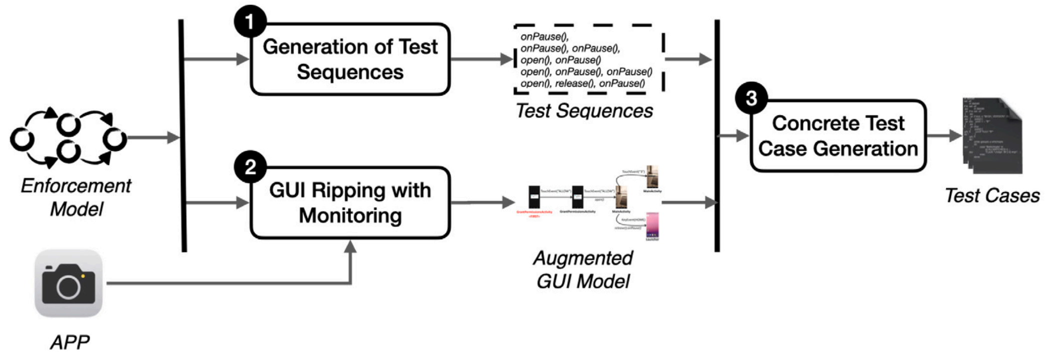
1
Generation of Test Sequences
onPause(),
onPause(), onPause(),
open(), onPause()
open(), onPause(), onPause()
open(), release(), onPause()
Test Sequences
3
Concrete Test Case Generation
Test Cases
Enforcement Model
2
GUI Ripping with Monitoring
Augmented GUI Model
APP

Fig. 4. The steps of the Test4Enforcers approach.

Fig. 4 provides a high-level overview of how Test4Enforcers generate test cases. Given an enforcement model denoted as an edit automaton and an app, the approach generates test cases aimed at validating the behavior of a software enforcer implementing the model when applied to the app. Notably, it is important to highlight that the approach does not need the actual software enforcer as an input to generate the tests. Instead, it leverages the specification of the enforcement strategy (i.e., the enforcement model) to identify the behaviors that have to be validated, and it explores the capability to interact with the app under testing to assess how to indirectly trigger the execution of the enforcer. The availability of the actual software enforcer is only required when the generated tests are *executed*.

The Test4Enforcers approach consists of three main steps, with the first step being the *Generation of Test Sequences*. In this step, the focus is on identifying a set of test sequences that effectively cover the behaviors specified in the enforcement model. These test sequences consist of method calls from the enforcement model and serve as the basis for comprehensive testing of the software enforcer.

However, it is important to note that at this stage, Test4Enforcers does not have the knowledge about how to exercise the app in a way that guarantees the coverage of these test sequences. To address this challenge, Test4Enforcers introduces the second step, namely *GUI Ripping with Monitoring*. In this step, Test4Enforcers applies a systematic process called GUI Ripping (Memon et al., 2013) to explore the User Interface (UI) of the target app. During this exploration, relevant events (method calls) related to the behavior of the enforcer are logged using the Android Profiler (Android Docs, 2023g). This GUI Ripping process helps identifying which GUI interactions within the app can potentially trigger the method calls present in the test sequences.

This step outputs the *Augmented GUI Model*, a finite state model that encapsulates the explored GUI states, UI interactions leading to state transitions, and events from the alphabet of the enforcement model (method calls) generated during these transitions. It essentially bridges the gap between the abstract test sequences and the concrete actions needed to exercise the app effectively.

It is important to note that, although we describe these steps sequentially, the first and second steps can occur in parallel. Finally, in the third step, namely *Generation of Concrete Test Cases*, the focus shifts to identifying within the Augmented GUI Model the sequences of UI interactions that precisely cover the previously generated test sequences (method calls from the enforcement model). These interactions, enriched with program oracles,[6] constitute the *Concrete Test Cases* that can be executed to validate the functionality of the software enforcer. In summary, these three steps are the core of the Test4Enforcers approach, enabling comprehensive testing and validation of software enforcers. In the following, we describe each step more rigorously.

4.1. Generation of test sequences

In this step, Test4Enforcers identifies the sequences of operations that must be covered with testing based on the behavior specified in the enforcement model. A notable method to generate these sequences from finite state models is the *W-method* (Chow, 1978; Lee and Yannakakis, 1996; Sidhu and Leung, 1989).

The *W-method* has been designed to unveil possible faults in implementations, such as, erroneous next-states, extra/missing states, etc. The main idea of the method is that starting from a state-based representation of a system, it is possible to generate sequences of inputs to reach every state in the model, cover all the transitions from every state, and identify all destination states to ensure that their counterparts in the implementation are correct. Since in our case the model represents the behavior of the enforcer, the application of the method would lead to a test suite that can thoroughly exercise any software enforcer implementing the enforcement model used to generate the test cases.

One limitation of the W-method is that it requires the model of the system to be *completely specified*, that is, in every state of the model there must be a transition for every possible input. This is however not the case for enforcers, since the inputs are generated by other components and frameworks, and not all the combinations of inputs can be feasibly generated (e.g., `onPause()` and `onResume()` are callbacks produced when an activity is paused and resumed, respectively, as a consequence it is impossible to produce a sequence of two `onPause()` events without an intermediate `onResume()` event). In order to tackle this limitation, we use the *Harmonized State Identifiers* (HSI) method, which is a variant of the W-method that does not require the model of the system to be completely specified (Luo et al., 1995; Belli et al., 2015). HSI exploits a few key concepts that are introduced below and that we straightforwardly applied to edit automata referring to the Ω and λ functions.

To generate an effective test suite, it is important to be able to distinguish the covered states. Given an edit automaton $A = \langle \Sigma, S, s_0, \delta \rangle$, its input function Ω and its output function λ, we say that two states $s_i, s_j \in S$ are *distinguishable* if there exists a *separating sequence* $\gamma \in \Omega(s_i) \cap \Omega(s_j)$, such that $\lambda(s_i, \gamma) \neq \lambda(s_j, \gamma)$, otherwise they are not distinguishable. In other words, states are distinguishable if there is a sequence of inputs that leads to different output sequences in these states.

To generate a thorough test suite, HSI exploits the notions of *transition cover* and *separating families*. We say that the set TC is a *transition cover* of A if for each transition x from state s there exists the sequence $\alpha x \in TC$ such that $\alpha \in \Omega(s_0)$ and s is the state reached by accepting α. By definition, TC also includes ϵ, where ϵ represents the empty sequence.

For instance, a transition cover TC for the enforcer in Fig. 2 is given by the following set

$\epsilon,$
$activity.onPause()_{req},$
$camera.open()_{req},$
$camera.open()_{req}\ activity.onPause()_{req},$
$camera.open()_{req}\ camera.release()_{req}$

The sequences in the transition cover, each one defined to cover a different transition, are extended with actions aimed at determining if the right state has been finally reached once the sequence is executed. To this end, HSI computes the *separating families*, which are sets of input sequences, one set for each state, that can be executed to distinguish a state from the other states of the system. In particular, a separating family is a set of input sequences $H_i \subseteq \Omega(s_i)$ for $s_i \in S$ satisfying the following condition: For any two distinguishable states s_i, s_j there exist sequences $\beta \in H_i, \gamma \in H_j$, such that α is a common prefix of β and γ and $\lambda(s_i, \alpha) \neq \lambda(s_j, \alpha)$. That is, H_i includes enough input sequences to distinguish the state s_i from any other state in the model once executed.

Computing the separating families for the automaton in Fig. 2 is straightforward since the two states have a single input in common that produces different outputs, allowing to distinguish the states. Thus $H_0 = H_1 = \{activity.onPause()_{req}\}$.

The HSI method can take into consideration the case the actual states of the implementation differ from the number of states in the model. However, we expect the software enforcer to have exactly the same number of states reported in the model, since the core implementation of the enforcer is normally generated automatically (Luo et al., 2014; Riganelli et al., 2019). In such a case, the resulting test sequences are obtained by concatenating the transition coverage set TC with the separating families H_i. Note that the concatenation considers the state reached at the end of each sequence in TC, namely s_i, to concatenate such a sequence with the sequences in the corresponding separating family, namely H_i.

In our example, this process generates the following sequences to be covered with test cases:

$activity.onPause()_{req},$
$activity.onPause()_{req}\ activity.onPause()_{req},$
$camera.open()_{req}\ activity.onPause()_{req},$
$camera.open()_{req}\ activity.onPause()_{req}\ activity.onPause()_{req},$
$camera.open()_{req}\ camera.release()_{req}\ activity.onPause()_{req}$

[6] An oracle is a mechanism or criterion used to determine whether a software program's output matches the expected behavior for a given set of inputs.

The HSI method usually includes a step to remove duplicates and prefixes from the generated set. Test4Enforcers only removes duplicates. In fact, removing a sequence that is a prefix of another sequence may drop a feasible test sequence in favor of an infeasible one (e.g., the longer sequence might be impossible to generate due to constraints in the environment not represented in the enforcement model, while it might be still possible to test the shorter sequence). In our example, since the list includes no duplicates, it is also the set of sequences that Test4Enforcer aims to cover to assess the correctness of the enforcer in Fig. 2.

In summary, the identified set of sequences represents the scenarios that Test4Enforcers aims to cover. These sequences are critical to conducting a thorough evaluation of the enforcer implementation, ensuring that the enforcer is effectively tested over a diverse range of states and transitions, as identified by the HSI method. The identified sequences, derived from both the coverage of transitions and the separation families, contribute to a comprehensive evaluation of the correctness and robustness of the enforcer implementation, which, however, can only be tested by interacting with the GUI of the apps that are monitored by the enforcer. For this reason, in the next step we have to build a model of the GUI of the apps that is then used to derive executable GUI actions that can cover the test sequences.

4.2. GUI ripping augmented with monitoring

GUI Ripping is an exploration strategy that can be used to explore the GUI of an application under test with the purpose of building a state-based representation of its behavior (Memon et al., 2013). In particular, GUI ripping generates the state-based model of the app by systematically executing every possible action on every state encountered during the exploration, until a given time or action budget expires. Our implementation of Test4Enforcers targets Android apps and uses DroidBot (Li et al., 2017) configured to execute actions in a breadth-first manner to build the state-based model.

To manage the potential infinite number of sequences and ensure effective exploration, we define a maximum number of user interactions or actions that DroidBot can execute during the ripping process. Once the tool reaches this threshold, it will stop the ripping process, regardless of whether it has explored the entire GUI.

The model represents each state of the app GUI according to its set of visible views and their properties. More rigorously, a state of the app GUI $s \in S_{app}$ is a set of views $s = \{v_i | i = 1 \dots n\}$, and each view is defined by a set of properties $v_i = \{p_{i1}, \dots, p_{ik}\}$, where a property p_{ij} is a key–value pair (k, v). For instance, an `EditText` is an Android view that allows the users to enter some text in the app. The `EditText` has a number of properties, such as `clickable`, which indicates if the view reacts to click events and whose values could be either True or False, and `text`, which represents the text present in the view and whose values is exactly the String with the text in the view.

Operations that change the set of visible views (e.g., because an activity is closed and another one is opened) or the properties of the views (e.g., because some text is entered in an input field) change the state of the app. DroidBot uses the following set of actions A_{app} during GUI ripping: *touch* and *long touch*, which execute a tap and a long tap on a clickable view, respectively; *setText*, which enters a pre-defined text inside an editable view; *keyEvent*, which presses a navigation button; and *scroll*, which scrolls the current window.

The actual state-based representation of the GUI execution space of an app produced by GUI Ripping consists of the visited states and the executed actions. Test4Enforcers extends the model generated by GUI ripping by adding the information generated by the monitor, that is, the list of the relevant internal events (i.e., the events in the alphabet of the enforcer) executed during each transition. The state-based model thus shows both the UI interactions that can be executed on the app, their effect on the state of the app, and the internal events that are activated when they are executed.

More formally, the model resulting from the GUI ripping phase is a tuple (S_{app}, s_0, T_{app}), where S_{app} is the set of visited states, $s_0 \in S_{app}$ is the initial state, $T_{app} \subseteq S_{app} \times A_{app} \times in^* \times S_{app}$ is a set of transitions $\langle s_1, a_{app}, \langle in_1, \dots, in_k \rangle, s_2 \rangle$, where s_1 and s_2 are the source and target states of the transition, respectively, a_{app} is the UI interaction that causes the transition, and $\langle in_1, \dots, in_k \rangle$ is a possibly empty sequence of internal events observed during the transition (note these events are exactly the input actions of the enforcer). The resulting model includes everything is needed to obtain the concrete test cases (i.e., the sequences of UI operations that must be performed on the app) that cover the test sequences derived with HSI (i.e., the sequences of input operations of the enforcer that must be generated). Fig. 5 shows an excerpt of the model obtained by running the ripping phase on the `Plumeria` app while considering the alphabet of the enforcer shown in Fig. 2. The `Plumeria` app is briefly introduced in Section 2. For simplicity, we represent the states with the screenshots of the app. The labels above transitions represent UI interactions, while the labels below transitions, when present, represent internal events collected by the monitor. For instance, when the *KeyEvent(Back)* UI interaction is executed and the app moves from the state *MainActivity* to the state *Launcher*, the sequence of internal events `camera.release()` `activity.onPause()` is observed.

Note that Test4Enforcers assumes that the software under test has a mostly deterministic behavior, that is, an action performed on a given state always produces the same computation, which is often the case for Android apps. When this is not the case, the test case generation algorithm presented in the next section can address non-determinism by trying multiple sequences of UI operations, until finally emitting the desired sequence of internal events.

4.3. Generation of concrete test cases

Test4Enforcers utilizes the GUI state-based model of the apps to create specific and executable test cases. These test cases are derived to cover the test sequences obtained through the HSI method. The goal is to identify sequences of GUI actions that trigger the execution of the desired sequences of internal events. The generation of concrete test cases is crucial for conducting thorough testing of the enforcer. This is particularly important because the enforcer cannot be effectively tested in isolation and must be evaluated in conjunction with monitored apps within the target environment.

Algorithm 1 outlines the test suite generation process. It takes an app, its GUI model, and a set of test sequences to be covered as input, and returns a test suite that includes executable GUI test cases covering the identified test sequences of internal method calls.

Algorithm 1 starts by initializing the test suite to the empty set (line 1), then for each sequence in the set of test sequences to be covered, the algorithm searches for a test (i.e., a path in the Augmented GUI model) that covers the sequence (`for` loop starting at line 2), and, if successful, it both adds an oracle to the test and adds the test to the test suite (lines 8 and 9, respectively). To identify the concrete test case that can cover a sequence, the algorithm searches for one or more paths in the model that exercise the desired sequence of events (line 3). Specifically, it iterates over transitions in a breadth-first search starting from the initial state and finds paths that cover the required sequence of internal events.

For instance, if the sequence to be covered is `camera.open()` `camera.release()` `activity.onPause()` and the GUI model is the one in Fig. 5, the algorithm can derive the sequence *TouchEvent(v1) TouchEvent(v2) KeyEvent(BACK)* as the concrete test to execute. In fact, the execution of the identified UI events is expected to produce the desired computation (based on the labels on the transitions). Since an arbitrarily large number of paths covering the desired sequence can be often determined, and not necessarily any path will deterministically produce the desired set of events internal to the app, the algorithm identifies the N (e.g., N=10) shortest paths of the model that cover

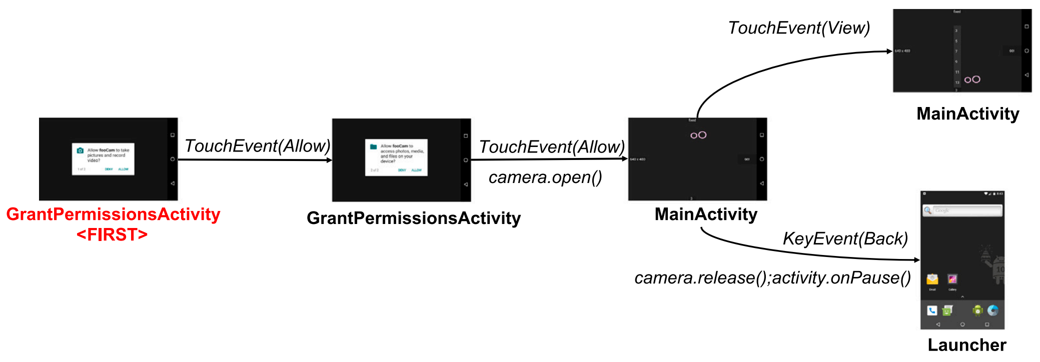

Fig. 5. Excerpt of a model derived with GUI Ripping.

the sequence (stored in variable *testCases* in the algorithm). Each path of UI events is tested by actually running the application to make sure that both its execution is feasible and it indeed generates the intended sequence of events (`for` loop at line 5). Our implementation of Test4Enforcers uses again DroidBot to reproduce a sequence of UI events.

Algorithm 1: Algorithm for generating concrete test cases

```
Input: App app, RippingModel appModel, TestSequences testSequences
Output: Set of <TestCase, TestSequences> testSuite
1  $testSuite \leftarrow \emptyset$
2  foreach $ts \in testSequences$ do
3      $testCases \leftarrow$ generateEventSequences($ts$, $appModel$)
4      $isCovered \leftarrow$ FALSE
5      foreach $tc \in testCases \wedge \neg isCovered$ do
6          $isCovered \leftarrow$ runTestCase($tc$, $ts$, $app$)
7          if $isCovered$ then
8              tc $\leftarrow addOracle(tc, ts)$
9              $testSuite$.add($tc$, $ts$)
10 return testSuite
```

If the right test is found, the algorithm embeds a *differential oracle* in the test case, before adding it to the test suite. A differential oracle is an oracle that determines the correctness of a test execution by comparing two executions of the same tests on two different programs. In our case, the compared programs are the app *with* and *without* the enforcer deployed. Test4Enforcers can inject two different differential oracles, depending on the characteristics of the sequence of events $in_1, \dots, in_k$ covered by the test $tc = a_1, \dots, a_n$ where the oracle must be embedded: the transparent-enforcement oracle and the actual-enforcement oracle.

Transparent-Enforcement Oracle. If the sequence is not the result of any change performed by the enforcer, that is, it covers a path of the enforcement model where the inputs and the outputs are always the same, the test is annotated as a test that must *produce the same result if executed on both the application with and without the enforcer*. More rigorously, if the output $o_1, \dots, o_m$ is produced by the enforcement model for the inputs $in_1, \dots, in_k$, this oracle applies when $k = m$ and $in_i = o_i, \forall i = 1 \dots k$. The resulting oracle checks the correctness of the execution by first capturing the intermediate states traversed during test execution, as done during the construction of the GUI model, and comparing them when collected from the app with and without the enforcer deployed. More rigorously, if the states cs_i and cs_i' are the states reached after executing the action a_i on the app without and with the enforcer, respectively, the oracle checks if $cs_i = cs_i', \forall i = 1 \dots n$. If the check fails, the enforcer *is unexpectedly intrusive*, although it was not supposed to be, and a failure is reported. For instance, the input sequence `camera.open()` `camera.release()` `activity.onPause()` is not altered by the enforcer in Fig. 2 and thus the transparent-enforcement oracle is used to determine the correctness of the test that covers this sequence, that is, no behavioral difference must be observed when this sequence is executed in both the app without and the app with the enforcer.

Actual-Enforcement Oracle. If the tested sequence corresponds to a path that requires the intervention of the enforcer, the test is annotated as producing an execution that may produce a different outcome when executed on the app with and without the enforcer in place. In such a case, given a sequence of events $in_1, \dots, in_k$, $\exists v, s.t.\ in_i = o_i \forall i < v$ and $in_v \neq o_v$. The resulting oracle checks the equality of the states visited by the test case executed on the app with and without the enforcer until the event in_v is produced, and checks for the possible presence of a difference in the follow-up states. More rigorously, if a_v is the GUI action that generates event in_v, the actual-enforcement oracle first checks if $cs_i = cs_i', \forall i < v$. If the check fails, the enforcer is unexpectedly intrusive and a failure is reported. For the remaining portion of the execution, it is not possible to know a priori if the activity of the enforcer must result in an effect visible on the GUI. The actual-enforcement oracle thus looks for such a difference, and if the difference is not found, it only issues a warning, suggesting that the enforcer may have failed its activity. Formally, the oracle checks if $\exists p, s.t., cs_p \neq cs_p'$ with $p \geq v$, if it is not the case the warning is issued. For instance, the input sequence `camera.open()` `activity.onPause()` causes the intervention of the enforcer shown in Fig. 2, which outputs an extra event `camera.release()`. The test corresponding to that sequence is thus labeled as *producing the same result until the activity.onPause() event, and a potentially different result afterwards*, and the actual-enforcement oracle is embedded in the test.

5. Evaluation methodology

In this section, we evaluate the effectiveness and efficiency of Test4Enforcers addressing the two research questions presented in the introduction. We assess its effectiveness in detecting faults in software enforcers (RQ1) and measure its efficiency in terms of execution time when testing enforcers (RQ2).

The evaluation methodology, visually summarized in Fig. 6, consists of four steps. It starts with a software enforcer that has to be validated with Test4Enforcers, its associated enforcement model, and one or more apps that use the API targeted by the enforcement model. In the *Mutant Generation* step, we automatically seed faults into the software enforcer to generate multiple faulty implementations of the enforcer that are used to assess the effectiveness and efficiency of the test cases generated with Test4Enforcers. In the *App Variant Generation* step, we produce a variant for each input app. If the original app satisfies the correctness policy enforced by the model, we will modify its variant to make it violate the policy, and vice-versa. In this way, we obtain a set of pairs of apps, where each pair consists of a same app in two different versions: a *correct* app that satisfies the policy and a *faulty* app that violates the policy. The *Test4Enforcers* step involves the use of Test4Enforcers to generate a comprehensive set of test cases. These test cases can

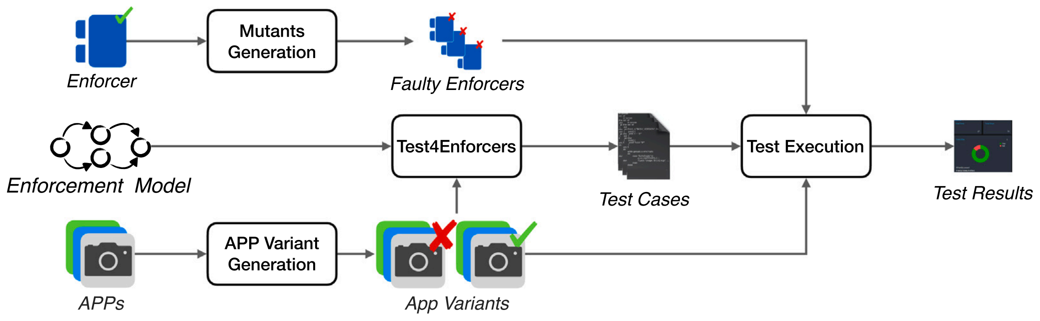

Fig. 6. Evaluation methodology.

validate the software enforcer within the context of the provided apps, ensuring thorough testing under both compliant (correct apps) and non-compliant (faulty app) scenarios. Finally, in the *Test Execution* step, the generated test cases are executed on the app variants, both with and without the activation of the faulty enforcers. We collect data about both the effectiveness and efficiency to answer RQ1 and RQ2:

- To answer RQ1 about the effectiveness of Test4Enforcers in detecting faults in the enforcer under test, we use mutation testing (Jia and Harman, 2011) to assess how well the tool identifies and flags faults introduced into the enforcer's source code through a set of carefully selected mutation operators.
- To address RQ2 about the efficiency of Test4Enforcers, we collect measurements to determine the time taken by Test4Enforcers to complete the enforcer testing process for a given enforcer.

In the following sections, we describe the policies (Section 5.1), the enforcers (Section 5.2), the apps (Section 5.3), the mutants (Section 5.4), and the testbed (Section 5.5) used for the evaluation. The empirical results are presented in Section 6.

5.1. Policies

To evaluate the effectiveness of Test4Enforcers, we first identified a set of API usage policies and we then defined the corresponding enforcement models. To identify the policies, we exploited the information about the misuses of Android APIs reported in Riganelli et al. (2019), Liu et al. (2019, 2016). As a result, we identified a total of 13 Android API usage policies, which are reported in Table 1. Column *ID* contains an identifier of the policy, in the rest of the paper this identifier is also used to identify the associated enforcer. Column *Lifecycle Object* indicates the Android component that has to interact with the API specified in column *API* according to the policy. Column *Correctness Policy* indicates the policy that specifies how to use the API.

The policies that we found correspond to three possible patterns that we report consistently to ease the understanding of the work. That is, we have written the policies in the forms “if `methodA` is invoked, invoke `methodB` when `callback`”, “if `methodA` is invoked, replace it with `methodB`”, and “if `methodA` is invoked, do not invoke `methodB`”. The first policy must be interpreted as: if the app invokes `methodA`, it must also invoke `methodB` when `callback` is produced by the Android framework, unless `methodB` has been already invoked before. This policy requires interrupting any ongoing usage of an API if certain events occur, such as the suspension or destruction of the current activity. The second policy must be interpreted as: if the app invokes `methodA`, the call must be replaced with a call to `methodB`. This policy requires replacing calls to faulty or deprecated methods with calls to methods that properly implement the required logic. The third policy must be interpreted as: if the app invokes `methodB` after `methodA`, suppress the call to `methodB`.

5.2. Enforcers

For each policy, we define an *enforcement model* that rigorously defines how to react to any violation of the correctness policies. The objective of the enforcement model is to *enforce* a correctness policy, when the running app does not satisfy it. We defined the enforcement model uniquely using the knowledge of the library API and the Android lifecycle events (i.e., the callback methods) (Riganelli et al., 2018; Android Docs, 2023d,f,c,e), which are the same for any app. Thus its definition does not require any knowledge specific to the app that uses the library. In particular, we used the enforcement models defined in Riganelli et al. (2019).

To obtain the software enforcers corresponding to the defined enforcement model for our experimental evaluation, we used the model-driven software development toolchain described in Riganelli et al. (2019), where we specified the enforcement model and automatically turned the specification into a Xposed-based (XDA, 2023) Java implementation of the software enforcer.

To ensure the correctness of the generated enforcers we manually inspected the generated code, in addition to running it on multiple apps of our benchmark. Fig. 7 shows the size of the generated software enforcers, which ranged from 559 lines of code for the policy P09 (`WifiMulticastLock` API) to 2K lines of code for the policy P11 (`MediaPlayer` API).

5.3. Apps

In order to execute and test the software enforcers, we need apps that interact with the API referred to in the enforcement policy, so that the enforcers can be added to the apps and their effect assessed. Thus, an app is a valid subject for our evaluation if it either violates or satisfies one or more of the identified policies. Although Test4Enforcers does not require app source code, we selected open source apps to manually verify the API usage protocol and generate app variants. In particular, we evaluated the proposed approach on 9 open-source applications randomly selected from GitHub among the ones that actually use of the API methods specified in the enforcement models. Table 2 lists all the subject apps, reporting their name (column *Application Name*), the year of the last update (column *Last Update*), the GitHub link where the source code is available (column *Source Code*), and the list of policies covered by that application (column *Covered Policies*). The apps are from a variety of domains, including communication and productivity.

To obtain two variants, one using the policy correctly and one violating the policy, we manually modified the apps. This change is introduced to make sure to have two scenarios to be investigated for each app: one that activates the enforcer to change the execution (app that violates the policy), and one that requires the enforcer to monitor the activity without altering the execution (app that satisfies the policy). For instance, we consider the enforcement model in Fig. 2, and the sequence of events `camera.open()`, `camera.release()`, `activity.onPause()` emitted by a correct app, no enforcement

Table 1
Evaluated policies.

ID	Lifecycle object	API	Correctness policy
P01	`Activity`	`android.bluetooth.BluetoothAdapter`	If `enable()` is invoked, invoke `disable()` when `onDestroy()`
P02	`Activity`	`android.content.ContentResolver`	If `managedQuery()` is invoked, replace it with `query()` of ContentResolver
P03	`Activity`	`android.hardware.Camera`	If `open()` is invoked, invoke `release()` when `onPause()`
P04	`Activity`	`android.hardware.Camera`	If `startPreview()` is invoked, invoke `stopPreview()` when `onPause()`
P05	`Activity`	`android.location.LocationManager`	If `requestLocationUpdates()` is invoked, invoke `removeUpdates()` when `onPause()`
P06	`Service`	`android.location.LocationManager`	If `requestLocationUpdates()` is invoked, invoke `removeUpdates()` when `onDestroy()`
P07	`Activity`	`android.os.RemoteCallbackList`	If `register()` is invoked, invoke `unregister()` when `onPause()`
P08	`Activity`	`android.hardware.SensorManager`	If `registerListener()` is invoked, invoke `unregisterListener()` when `onPause()`
P09	`Activity`	`android.net.wifi.WifiManager.MulticastLock`	If `acquire()` is invoked, invoke `release()` when `onDestroy()`
P10	`Activity`	`android.util.LruCache`	If `<init>` is invoked, invoke `evictAll()` when `onDestroy()`
P11	`Activity`	`android.media.MediaPlayer`	If `create()` is invoked, invoke `release()` when `onPause()`
P12	`MediaRecorder`	`android.hardware.Camera`	If `MediaRecorder.start()` is invoked, do not invoke `lock()`
P13	`Activity`	`android.os.PowerManager.WakeLock`	If `acquire()` is invoked, invoke `release()` when `onPause()`

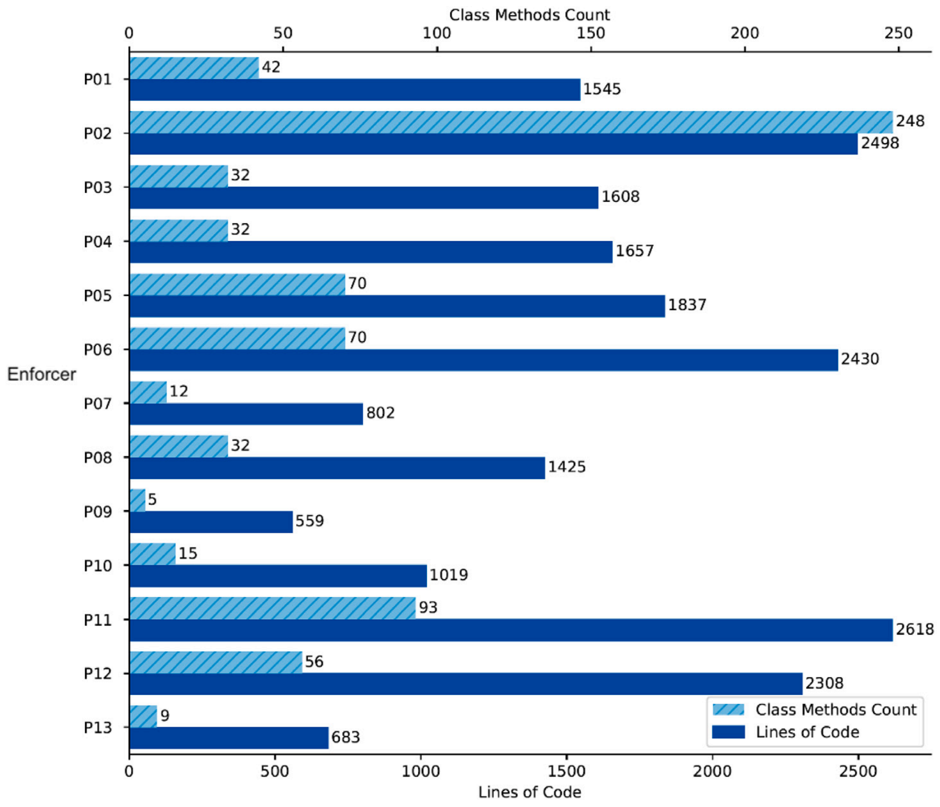

Fig. 7. Code size of the generated enforcers.

Table 2
Evaluated applications.

Application name	Last update	Source code	Covered policies
BlueChat	2016	https://github.com/AlexKang/blue-chat	P01
Contact List	2019	https://github.com/Bennyhwanggggg/Small-Android-Apps	P02
FooCam	2017	https://github.com/phunehehe/foocam	P03, P04
GetBack GPS	2022	https://github.com/ruleant/getback_gps	P05, P06, P07, P08
Multicast Tester	2017	https://github.com/MitchTalmadge/Multicast-Tester	P09
Pop Movies	2017	https://github.com/heiho1/popmovie	P10
Sample Music Player	2019	https://github.com/retzionibmhack/SampleMusicPlayer	P11
Video Recorder Camera	2017	https://github.com/anthorlop/VideoRecorderCamera	P12
Wakelock Revamp	2021	https://github.com/d4rken/wakelock-revamp	P13

action is needed because the camera would be released properly when the application is closed. In contrast, if we consider the same enforcement model in Fig. 2, but we instead consider the sequence of events `camera.open()`, `activity.onPause()` emitted by a faulty app, the enforcer will have to intervene to change the execution.

When we find an app that violates a policy, we simply obtain the correct one by fixing the app. For the correct apps, we modified the interaction with the API to introduce a fault. For instance, if the policy states: "If `open()` is invoked, invoke `release()` when `onPause()`" we introduced a fault by removing the call to the `release()` method of the `Camera` instance in the `onPause` lifecycle event, simulating the case of a developer who forgot to release an acquired camera. Table 3 summarizes the changes applied to applications by specifying the app name (column *Application name*), the policy identifier associated with the code change (column *Policy*), the class that was changed (column *Class*), the method that was altered (column *Method*), the lines of code involved in the change (column *Line(s)*), the type of change (column *Type*), which can be FIX if it fixes a policy violation or FAULT if it introduces a policy violation, and a brief sentence about the applied change (column *Description*).

The set of apps obtained in such a way enables the evaluation of the tests generated by Test4Enforcers both when the enforcer must alter the execution of the app to satisfy a policy and when the enforcer must not alter the execution because the app correctly implements the policy. If we consider the enforcement model in Fig. 2, and the sequence of events `camera.open()`, `camera.release()`, `activity.onPause()`, no enforcement action is needed because the camera is released properly when the application is closed, thus this sequence can be covered only using apps that *satisfy the policy*. In contrast, the apps that *do not satisfy the policy* can be used to cover the test sequences where the enforcer must modify the execution so as not to violate the policy. For instance, if we consider the enforcement model in Fig. 2, the sequence of events `camera.open()`, `activity.onPause()` can be covered only using an app that does not satisfy the policy.

5.4. Mutants

To evaluate the fault detection capability of Test4Enforcers, we injected mutation faults into the generated enforcers to create a set of faulty enforcers. We use Major and its full set of mutation operators to automatically introduce mutations in software enforcers (Just, 2014). We filtered out the software enforcers that do not compile after the mutation, or that include mutations that do not affect the semantics of the enforcer leading to a software enforcer that is functionally *equivalent* to the original one. In particular, we manually identified and removed mutants belonging to two classes:

- Equivalent mutants: these mutants are syntactically different but semantically equivalent to the original enforcer and therefore cannot be killed by any test case. For example, changing if $x > 0$ to $x \geq 0$ when x cannot be 0, or adding a mutation to the logging code have no impact on the semantics of the enforcer.
- Mutants in non-alphabet methods: we focus on detecting defects in the code that implements the enforcement behavior. For this reason, we discarded mutants that modified the behavior of methods that were not in the alphabet of the enforcment model.

We refer to the set of mutated enforcers that compile and are not equivalent to the original enforcer as the *valid faulty enforcers*. Table 4 shows the mutation operators supported by Major that we used in our evaluation. These mutation operators are commonly applied to evaluate the effectiveness of a test suite (Siami Namin et al., 2008).

To assess the effectiveness of Test4Enforcers, we compute the *Mutation Score* (MS), that is the rate of valid faulty enforcers that are detected to be wrong by the test cases automatically generated by the tool:

$$MS = \frac{Detected\ Faulty\ Enforcers}{Valid\ Faulty\ Enforcers} \quad (1)$$

Table 5 displays the number of mutants through different phases of the mutation testing approach: the total number of produced mutants prior to the application of any mutants reduction techniques (column *Generated*), the remaining amount of mutants after the reduction process (column *Valid*), the final number of compiled mutants (column *Compiled*) and the breakdown of the compiled mutants for each available mutation operator (columns *AOR*, *COR*, *EVR*, *LVR*, *ROR*, *STD*). Mutation operators such as LOR (Logical), SOR (Shift), ORU (Unary) are not present in Table 5 because no suitable instructions were present in any enforcer source, thus no mutants was produced from such mutations.

5.5. Testbed

The experiments have been conducted with the Genymotion v3.2.1 Android emulator using an emulated device equipped with Android 7.1 (API 25) and 3 GB of RAM. The only exception is due to limitations on the emulation of Bluetooth and positional sensors (e.g., gyroscope and accelerometer). The policies P01 and P08 that concern with this kind of sensors have been tested on a physical device: a Samsung Galaxy S5 running Android 6.0.1 (API 23), equipped with a 2.5 GHz quad-core Snapdragon 801 processor, 2 GB of RAM, and 16 GB of internal storage.

6. Empirical results

RQ1 - Can Test4Enforcers effectively detect faults in the enforcer under test?

To answer this question we generated the test cases with Test4Enforcers, and checked the mutants detected by the tests, to finally compute the mutation score. Table 6 reports the results. Column *Enforcer* indicates the identifier of the policy enforced by the enforcer. Column *Total* is the number of mutants (i.e., faulty enforcers) generated from the original software enforcer. Column *Detected* indicates the number of mutants that have been detected as faulty by Test4Enforcers. Column *Equivalent* indicates the number of mutants classified as equivalent to the original software enforcer and Column *MS* reports the achieved mutation score, according to Eq. (1). Finally, column *Crashes* reports the total number of mutants that results in app crashes.

Table 3
Selected applications.

Application	Policy	Class	Method	Line(s)	Type	Description
BlueChat	P01	`com.alexkang.bluechat.ClientActivity`	`onDestroy`	199	FIX	Add `adapter.disable()` call when `ClientActivity` is destroyed.
Contact List	P02	`benny.dev.contactsdatabaseapp.MainActivity`	`onClick`	74-80	FIX	Replace `managedQuery()` call with `contentResolver.query()` when the load contacts button is pushed.
FooCam	P03	`net.phunehehe.foocam.CameraPreview`	`surfaceDestroyed`	32-34	FIX	Add `camera.stopPreview()` call when the application is closed or the previewed camera is swapped (for instance, switching between front and rear cameras).
	P04	`net.phunehehe.foocam.MainActivity`	`releaseCamera`	172	FAULT	Remove `camera.release()` call when the application is closed.
GetBack GPS	P05	`com.github.ruleant.getback_gps.LocationService`	`onDestroy`	187	FAULT	Remove `mLocationManager.removeUpdates (mListener)` call when the application is closed.
	P06	`com.github.ruleant.getback_gps.LocationService`	`onDestroy`	187	FAULT	Remove `mLocationManager.removeUpdates (mListener)` call when the service is unbound.
	P07	`com.github.ruleant.getback_gps.LocationService`	`onDestroy`	181-183	FIX	Add `mCallbacks.unregister(callback)` call on each registered callback when the application is closed.
	P08	`com.github.ruleant.getback_gps.LocationService`	`onDestroy`	190	FAULT	Remove `mSensorOrientation.removeEvent-Listener(this)` call when the application is closed.
Multicast Tester	P09	`net.liveforcode.multicasttester.MainActivity`	`setWifiLockAcquired`	185;194	FAULT	Remove `wifiLock.release()` call when the application is closed.
Pop Movies	P10	`com.example.heiho1.popmovies.service.CachingService`	`onDestroy`	41	FAULT	Remove `cache.evictAll()` call when the application is closed.
Sample Music Player	P11	`com.revosleap.samplemusicplayer.playback.MediaPlayerHolder`	`release`	325	FAULT	Remove `mMediaPlayer.release()` call when the application is closed or the song is changed.
Video Recorder Camera	P12	`es.anthorlop.camera.activities.CameraActivity`	`onClick`	448	FAULT	Add `mPreview.getCamera().lock()` call after the video recording has started.
Wakelock Revamp	P13	`eu.thedarken.wldonate.main.core.locks.PartialWakeLock`	`release`	32	FAULT	Remove `lock.release()` call when the application is closed.

Table 4
Mutation operators implemented in Major.

Mutation operator	Example
AOR (*A*rithmetic *O*perator *R*eplacement)	`a + b` ⟼ `a - b`
LOR (*L*ogical *O*perator *R*eplacement)	`a ∧ b` ⟼ `a | b`
COR (*C*onditional *O*perator *R*eplacement)	`a && b` ⟼ `a || b`
ROR (*R*elational *O*perator *R*eplacement)	`a == b` ⟼ `a >= b`
SOR (*S*hift *O*perator *R*eplacement)	`a >> b` ⟼ `a << b`
ORU (*O*perator *R*eplacement *U*nary)	`-a` ⟼ `~a`
EVR (*E*xpression *V*alue *R*eplacement)	`return a` ⟼ `return 0`
LVR (*L*iteral *V*alue *R*eplacement)	`0` ⟼ `1`
STD (*ST*atement *D*eletion)	`foo(a,b)` ⟼ `<NO-OP>`

Table 5
Number of mutants at different stages of the generation process.

Enforcer	Mutants			Mutation operator					
	Generated	Valid	Compiled	AOR	COR	EVR	LVR	ROR	STD
P01	1121	282	255	4	108	34	23	37	49
P02	1874	147	133	4	36	24	24	11	34
P03	1160	243	218	4	84	34	22	30	44
P04	1230	262	259	4	110	34	24	37	50
P05	1492	285	259	4	102	34	34	34	51
P06	2255	313	286	4	102	37	40	46	57
P07	584	277	247	4	102	34	26	34	47
P08	1154	285	260	4	100	34	38	34	50
P09	410	285	258	4	110	34	23	37	50
P10	729	277	251	4	93	34	28	36	54
P11	1913	239	215	4	80	32	31	26	44
P12	1927	265	237	4	80	36	30	42	45
P13	502	284	257	4	110	34	22	37	50
TOT	16 351	3443	3135	52	1217	435	365	441	625

Table 6
Mutants detected by Test4Enforcers.

Enforcer	Total	Detected	Equivalent	MS	Crashes
P01	255	88	137	74.58%	21
P02	133	47	68	72.31%	43
P03	218	71	98	59.17%	3
P04	259	87	133	69.05%	14
P05	259	66	107	43.42%	2
P06	286	79	131	50.97%	2
P07	247	83	94	54.25%	4
P08	260	70	110	46.67%	3
P09	258	69	91	41.32%	2
P10	251	113	89	82.10%	4
P11	215	86	72	60.14%	0
P12	237	75	104	56.39%	2
P13	257	78	122	57.78%	17
TOT	**3135**	**1012**	**1356**	**63.88%**	**117**

The results show that Test4Enforcers can effectively reveal faults in software enforcers. In fact, Test4Enforcers automatically detected from 41.32% (WifiMulticastLock) to 82.10% (LruCache) of the faulty enforcers, with an overall mutation score equals to 56.89%.

Interestingly, we noticed that most of the unidentified faulty mutants exhibit their faulty behavior only in corner cases. For example, consider a mutation that removes the reset of an internal flag indicating whether or not a relevant policy method has already been handled during runtime enforcement activity. The first time that method is accessed, the enforcer will act correctly because the flag is initialized with the correct value for execution. However, during subsequent invocations, the behavior will be wrong since the flag will prevent the correct execution of the enforcement actions.

The number of faults resulting in crashes shows the upper bound of faults that can be discovered by state of the art tools relying on the detection of crashes and exceptions (Android Docs, 2023h; Alshahwan et al., 2018; Romdhana et al., 2022; Peng et al., 2022; Mariani et al., 2014). Interestingly, only 117 mutants were killed (5.5% mutation score) because they caused the application to crash.

RQ2 - What is the execution time of Test4Enforcers when testing enforcers?

To answer this question, we measured the time taken by Test4Enforcers in completing the test of an enforcer by using an app. Table 7 reports the total execution time in seconds (column *Testing Time*), for testing each enforcer identified by the policy id (column *Enforcer*).

Testing an enforcer might be potentially expensive, since the same sequences of operations must be executed twice: on the app with and without the enforcer. Moreover, the device must be reboot after every test to guarantee the lack of interference between the test cases. Rebooting is an expensive operation, especially when executed on the real device, as strictly required by policies P01 and P08. Despite these issues the overall testing time is still practical, in fact it varied from nearly 2 min to slightly less than 10 min. Moreover, although sometime expensive in terms of resources, the process does not require the manual intervention of the testers, being convenient in terms of human effort.

6.1. Threats to validity

In this subsection, we report the main factors that could potentially affect the validity of our findings.

Internal Validity. One concern regarding internal validity is about our automated mutant generation process. We rely on mutation operators provided by Major, which might not encompass all conceivable faults that real-world enforcers can encounter. To mitigate this threat we used a comprehensive set of mutation operators provided by Major that are widely recognized in the field.

Table 7
Total duration of each experiment.

Enforcer	Testing time (s)
P01	562
P02	148
P03	201
P04	304
P05	167
P06	184
P07	123
P08	469
P09	150
P10	179
P11	219
P12	109
P13	213

External Validity. While our study predominantly uses open-source Android applications to assess Test4Enforcers, we acknowledge the potential limitation in terms of generalizability. Open-source apps, while valuable, may not fully represent the diversity and complexity of every real-world application. The policies, enforcers, and mutants we used were carefully selected, but may not cover all possible scenarios. Although not fully generalizable, our findings are yet obtained by working with thousands of mutants, which is a fairly large mutation-based experiment. We plan to extend Test4Enforcers to a broader set of domains and policies in the future.

Construct Validity. Another concern relates to the construct validity of our study. A specific concern is the selection of the policies. Some crucial policies might have been inadvertently omitted, potentially affecting the comprehensiveness of our study. To address this potential construct validity concern, we have selected policies for evaluation based on prior research and known issues in Android API usage. However, it is important to recognize that these policies may not encompass the entire spectrum of policies relevant to Android app development, and additional work might be needed to obtain a better understanding of the most relevant policies.

Subjectivity and Human Judgment. Lastly, human judgment plays a pivotal role in our study, particularly in manually inspecting mutants and app variants. The subjective nature of identifying equivalent mutants or assessing app behavior introduces variability, as different individuals may perceive these aspects differently. This subjectivity could potentially introduce bias into our results, which is an important consideration when interpreting our findings. To mitigate this subjectivity, we applied clear guidelines and criteria for evaluating equivalence mutants and app variants, in addition to discussing the unclear cases among all the authors of the paper.

7. Related work

The contribution described in this paper spans four related research areas: runtime enforcement, automated testing, model-based testing, and verification of runtime enforcement.

Runtime enforcement solutions can be used to prevent a software system from behaving incorrectly with respect to a set of known policies. In particular, runtime enforcement strategies modify executions assuring that policies are satisfied despite the potentially incorrect behavior of the monitored software (Falcone, 2010; Khoury and Tawbi, 2012). Enforcement strategies can be specified using a variety of models, including models specifically designed to represent runtime enforcement, such as All or Nothing Automata (Bielova and Massacci, 2011), Late Automata (Bielova and Massacci, 2011), Mandatory Results Automata (Dolzhenko et al., 2015), and Edit Automata (Ligatti et al., 2005). Runtime enforcement solutions have been applied in multiple application domains, including mobile applications (Riganelli et al., 2016, 2017a; Falcone et al., 2012; Daian et al., 2015), operating systems (Sidiroglou et al., 2009), web-based applications (Magalhães and Silva, 2015), and cloud systems (Dai et al., 2009). An overview of techniques to prevent failures by enforcing the correct behavior at runtime has been recently published by Falcone et al. (2018). Among these many domains, in this paper we focus on the Android environment, which has been already considered in the work by Falcone et al. (2012), who studied how to enforce privacy policies by detecting and disabling suspicious method calls, and more recently by Riganelli et al. (2017a, 2019, 2018), who studied how to augment classic Android libraries with proactive mechanisms able to automatically suppress and insert API calls to enforce resource usage policies.

While runtime enforcement strategies focus on the definition of models and strategies to specify and implement the enforcers, Test4Enforcers is complemental to this effort, since it derives the test cases that should be executed on applications with and without the enforcers to verify the correctness of the implemented enforcer.

Automated testing is a method for automatically generating test cases, which mainly involves automatically choosing interesting inputs and checking whether running the software using the selected inputs reveals any bugs. Choosing inputs is challenging because of the large space of possible inputs to choose from, while checking is challenging because the software often lacks any kind of formal specification of the expected behavior. Several techniques have been studied over the years, such as random (Android Docs, 2023h), model-based (Pan et al., 2020), and search-based (Alshahwan et al., 2018). Although these techniques apply different strategies for input selection, these techniques are mainly limited to the identification of bugs that cause crashes or unhandled exceptions. Test4Enforcers overcomes this limitation by providing a sophisticated approach to discover more complex noncompliant behaviors that do not necessarily evolve into a crash or exception.

Model-based testing (MBT) refers to the automatic generation of a suite of test cases from models extracted from requirements (Dalal et al., 1999; Dias Neto et al., 2007). The purpose of the generated test suite is to determine whether an implementation is correct with respect to its specification. MBT approaches are often organized around three main steps (Utting et al., 2012): building the model, choosing the test selection criteria and building the test case specifications, and generating tests. MBT has been extensively used in the software safety domain, where conformance of the implementation with respect to the model is critical, as shown in the survey by Gurbuz and Tekinerdogan (2018). Test4Enforcers is also a MBT approach, in fact it uses a model, it defines a coverage criterion, and it generates the corresponding test cases.

A variety of models have been used to guide test case generation, including finite state machines, UML diagrams (statechart, class, activity, and others), and Z specifications (Dias Neto et al., 2007). Indeed, finite-state models are among the most used ones (Hierons and Turker, 2016). Interestingly, there are various methods to derive test cases from finite-state machines. For instance, the W (Chow, 1978), Wp (Fujiwara et al., 1991), UIO (Sabnani and Dahbura, 1988), DS (Gonenc, 1970), HSI (Petrenko et al., 1996), and the H (Dorofeeva et al., 2005) are well-know test derivation methods (Dorofeeva et al., 2010). Test4Enforcers exploits HSI due to the characteristics of the models used to represent the behavior of the enforcers. Furthermore, Test4Enforcers defines a strategy to produce the target sequences of events while interacting with the UI of an application.

Verification of runtime enforcement concerns with checking that the software enforcer is indeed delivering the intended behavior. In fact, although the enforcer is meant to correct the behavior of a monitored software, the enforcer itself might still be wrong and its activity might compromise the correctness of the system rather than improving it. A recent work in this direction is the one by Riganelli et al. (2017b) that provides a way to verify if the activity of multiple enforcers may interfere. The proposed analysis is however entirely based on the models, and the many problems that might be introduced by the actual software enforcers cannot be revealed with that approach. Test4Enforcers provides a complemental capability, that is, it can test if the implementation of the enforcer behaves as expected once injected in the target system.

8. Conclusions

Runtime enforcement is a useful technique to guarantee that certain correctness policies are satisfied by a running software application. However, specifying the enforcement strategies and implementing the corresponding software enforcers might be challenging. In particular, translating an enforcement model into a software enforcer might be difficult because of the significant adaptation and instrumentation effort required to close the gap between the abstraction of the models and the actual implementation, which must take under consideration the requirements of the target execution environment. Indeed, enforcers may easily introduce side effects in the attempt of modifying executions. These are well-known shortcomings of software enforcement solutions (Bielova and Massacci, 2011; Riganelli et al., 2017b).

To address these problems, this paper presents Test4Enforcers, a test case generation approach that can automatically derive a test suite that can be used to validate the correctness of the software enforcers derived from enforcement models. Test4Enforcers exploits the information present in the enforcement model to first derive a specification of the test cases that should be executed to validate the corresponding software enforcer, and then automatically turns the specification into concrete test cases. To validate the exercised behaviors, Test4Enforcers run the same tests on the apps augmented with and without the enforcer, comparing their behavior, and thus assessing that the impact of the enforcer on the execution is the expected one.

We developed Test4Enforcers with a primary focus on Android systems, as apps downloaded and installed by end-users on Android devices are often developed by unknown and potentially untrusted parties. In such systems, software enforcers represent a particularly relevant option to enhance security and reliability. While our initial implementation targets Android, Test4Enforcers is designed with broader applicability in mind and the approach can be adapted to other systems as needed.

Our evaluation, which involved injecting 3,135 faults into 13 software enforcers, demonstrates Test4Enforcers' effectiveness in identifying undesired enforcer behaviors. These findings directly address RQ1 (Effectiveness), showcasing Test4Enforcers' capability to detect faults and enhance enforcer reliability. Regarding RQ2 (Efficiency), we assessed Test4Enforcers' execution time for testing enforcers. Despite potential resource-intensive challenges, such as executing sequences of operations twice and device reboots, the overall testing time remained practical, ranging from approximately 2 to less than 10 min for different enforcers. Importantly, this process is automated, alleviating the need for manual intervention and minimizing human effort.

In our future work, we plan to expand Test4Enforcers in multiple directions. This includes extending Test4Enforcers to address additional classes of misbehaviors compared to the ones we considered in this paper, for instance using enforcement models that incorporate time information similarly to the ones considered in Falcone and Pinisetty (2019). We also intend to extend the evaluation of Test4Enforcers by considering a broader spectrum of faults and enhance cross-platform compatibility. Comparative studies and real-world deployments will help assessing its effectiveness and practicality in diverse contexts. These efforts aim to make Test4Enforcers more versatile and applicable in the development and reliability of enforcers.

CRediT authorship contribution statement

Oliviero Riganelli: Conceptualization, Validation, Writing – original draft, Writing – review & editing. **Daniela Micucci:** Conceptualization, Writing – original draft, Writing – review & editing. **Leonardo Mariani:** Conceptualization, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared the link to my data/code in the paper.

References

Alshahwan, N., Gao, X., Harman, M., Jia, Y., Mao, K., Mols, A., Tei, T., Zorin, I., 2018. Deploying search based software engineering with sapienz at facebook. In: Proceedings of the International Symposium Search-Based Software Engineering. SSBSE.

Android Docs, 2023a. Camera API. https://developer.android.com/guide/topics/media/camera.

Android Docs, 2023b. Camera API - Releasing the camera. https://developer.android.com/guide/topics/media/camera.html#release-camera.

Android Docs, 2023c. Fragment lifecycle. https://developer.android.com/guide/fragments/lifecycle.

Android Docs, 2023d. Handling Lifecycles with Lifecycle-Aware Component. https://developer.android.com/topic/libraries/architecture/lifecycle.

Android Docs, 2023e. Services overview. https://developer.android.com/guide/components/services.

Android Docs, 2023f. The activity lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle.

Android Docs, 2023g. The Android Profiler. https://developer.android.com/studio/profile/android-profiler.

Android Docs, 2023h. UI/Application exerciser monkey. https://developer.android.com/studio/test/other-testing-tools/monkey.

Belli, F., Beyazıt, M., Endo, A.T., Mathur, A., Simao, A., 2015. Fault domain-based testing in imperfect situations: A heuristic approach and case studies. Softw. Qual. J. 23 (3), 423–452.

Bielova, N., Massacci, F., 2011. Do you really mean what you actually enforced? Int. J. Inf. Secur. 10, 239–254.

Ceci, L., 2023. Number of mobile app downloads worldwide from 2016 to 2022. https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/.

Chow, T.S., 1978. Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. (3), 178–187.

Dai, Y., Xiang, Y., Zhang, G., 2009. Self-healing and hybrid diagnosis in cloud computing. In: Proceedings of the International Conference on Cloud Computing. CloudCom.

Daian, P., Falcone, Y., Meredith, P.O., Serbanuta, T., Shiriashi, S., Iwai, A., Rosu, G., 2015. RV-android: Efficient parametric android runtime verification, a brief tutorial. In: Proceedings of the International Conference on Runtime Verification. RV.

Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Patton, G.C., Horowitz, B.M., 1999. Model-based testing in practice. In: Proceedings of the International Conference on Software Engineering. ICSE.

Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H., 2007. A survey on model-based testing approaches: A systematic review. In: Proceedings of the ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies. WEASELTech.

Dolzhenko, E., Ligatti, J., Reddy, S., 2015. Modeling runtime enforcement with mandatory results automata. Int. J. Inf. Secur. 14 (1), 47–60.

Dorofeeva, R., El-Fakih, K., Maag, S., Cavalli, A.R., Yevtushenko, N., 2010. FSM-based conformance testing methods: A survey annotated with experimental evaluation. Inf. Softw. Technol. 52 (12), 1286–1297.

Dorofeeva, R., El-Fakih, K., Yevtushenko, N., 2005. An improved conformance testing method. In: Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems. FORTE.

Falcone, Y., 2010. You should better enforce than verify. In: Proceedings of the International Conference on Runtime Verification. RV.

Falcone, Y., Currea, S., Jaber, M., 2012. Runtime verification and enforcement for android applications with RV-Droid. In: Proceedings of the International Conference on Runtime Verification. RV.

Falcone, Y., Mariani, L., Rollet, A., Saha, S., 2018. In: Bartocci, E., Falcone, Y. (Eds.), Lectures on Runtime Verification: Introductory and Advanced Topics. Springer International Publishing, pp. 103–134, Ch. Runtime Failure Prevention and Reaction.

Falcone, Y., Mounier, L., Fernandez, J.-C., Richier, J.-L., 2011. Runtime enforcement monitors: Composition, synthesis, and enforcement abilities. Form. Methods Syst. Des. 38 (3), 223–262.

Falcone, Y., Pinisetty, S., 2019. On the runtime enforcement of timed properties. In: Proceedings of the International Conference on Runtime Verification. RV.

Fujiwara, S., v. Bochmann, G., Khendek, F., Amalou, M., Ghedamsi, A., 1991. Test selection based on finite state models. IEEE Trans. Softw. Eng. 17 (6), 591–603.

Gonenc, G., 1970. A method for the design of fault detection experiments. IEEE Trans. Comput. C-19 (6), 551–558.

Gurbuz, H.G., Tekinerdogan, B., 2018. Model-based testing for software safety: A systematic mapping study. Softw. Qual. J. 26 (4), 1327–1372.

Guzman, M., Riganelli, O., Micucci, D., Mariani, L., 2020. Test4Enforcers: Test case generation for software enforcers. In: Proceedings of the International Conference on Runtime Verification. RV.

Hierons, R.M., Turker, U.C., 2016. Parallel algorithms for generating harmonised state identifiers and characterising sets. IEEE Trans. Comput. 65 (11), 3370–3383.

Jia, Y., Harman, M., 2011. An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37 (5), 649–678. http://dx.doi.org/10.1109/TSE.2010.62.

Just, R., 2014. The major mutation framework: Efficient and scalable mutation analysis for java. In: Proceedings of the International Symposium on Software Testing and Analysis. ISSTA.

Khoury, R., Tawbi, N., 2012. Which security policies are enforceable by runtime monitors? A survey. Comp. Sci. Rev. 6 (1), 27–45.

Koetsier, J., 2020. There are now 8.9 million mobile apps, and China is 40% of mobile app spending. Forbes.

Könighofer, B., Bloem, R., Ehlers, R., Pek, C., 2022. Correct-by-construction runtime enforcement in AI – A survey. In: Raskin, J.-F., Chatterjee, K., Doyen, L., Majumdar, R. (Eds.), Principles of Systems Design. Springer Nature Switzerland, pp. 650–663.

Lee, D., Yannakakis, M., 1996. Principles and methods of testing finite state machines-A survey. Proc. IEEE 84 (8), 1090–1123.

Li, Y., Ziyue, Y., Yao, G., Xiangqun, C., 2017. DroidBot: A lightweight UI-guided test input generator for android. In: Proceedings of the International Conference on Software Engineering Companion. ICSE.

Ligatti, J., Bauer, L., Walker, D., 2005. Edit automata: Enforcement mechanisms for run-time security policies. Int. J. Inf. Secur. 4, 2–16.

Ligatti, J., Bauer, L., Walker, D., 2009. Run-time enforcement of nonsafety policies. ACM Trans. Inf. Syst. Secur. 12 (3).

Liu, Y., Wang, J., Wei, L., Xu, C., Cheung, S.-C., Wu, T., Yan, J., Zhang, J., 2019. Droidleaks: A comprehensive database of resource leaks in android apps. Empir. Softw. Eng. 24 (6), 3435–3483.

Liu, J., Wu, T., Yan, J., Zhang, J., 2016. Fixing resource leaks in android apps with light-weight static analysis and low-overhead instrumentation. In: Proceedings of the International Symposium on Software Reliability Engineering. ISSRE.

Luo, G., Petrenko, A., Bochmann, G.V., 1995. Selecting test sequences for partially-specified nondeterministic finite state machines. In: Proceedings of the IFIP WG 6.1 International Workshop on Protocol Text Systems.

Luo, Q., Zhang, Y., Lee, C., Jin, D., Meredith, P.O., Şerbănuţă, T.F., Roşu, G., 2014. RV-monitor: Efficient parametric runtime verification with simultaneous properties. In: Proceedings of the International Conference on Runtime Verification.

Magalhães, J.P., Silva, L.M., 2015. SHõWA: A self-healing framework for web-based applications. ACM Trans. Auton. Adapt. Syst. 10 (1), 4:1–4:28.

Mariani, L., Pezzè, M., Riganelli, O., Santoro, M., 2014. Automatic testing of GUI-based applications. Softw. Test. Verif. Reliab. 24 (5), 341–366.

Memon, A.M., Banerjee, I., Nguyen, B.N., Robbins, B., 2013. The first decade of GUI ripping: Extensions, applications, and broader impacts. In: Proceedings of the Working Conference on Reverse Engineering. WCRE.

Pan, M., Huang, A., Wang, G., Zhang, T., Li, X., 2020. Reinforcement learning based curiosity-driven testing of android applications. In: Proceedings of the International Symposium on Software Testing and Analysis. ISSTA.

Peng, C., Zhang, Z., Lv, Z., Yang, P., 2022. MUBot: Learning to test large-scale commercial android apps like a human. In: Proceedings of the International Conference on Software Maintenance and Evolution. ICSME.

Petrenko, A., Yevtushenko, N., v. Bochmann, G., 1996. Testing deterministic implementations from nondeterministic FSM specifications. In: Proceedings of the IFIP TC6 International Workshop on Testing of Communicating Systems.

Rasthofer, S., Arzt, S., Lovat, E., Bodden, E., 2014. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In: Proceedings of the International Conference on Availability, Reliability and Security. ARES, pp. 40–49.

Riganelli, O., Micucci, D., Mariani, L., 2016. Healing data loss problems in android apps. In: Proceedings of the International Workshop on Software Faults (IWSF), Co-Located with the International Symposium on Software Reliability Engineering. ISSRE.

Riganelli, O., Micucci, D., Mariani, L., 2017a. Policy enforcement with proactive libraries. In: Proceedings of the IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS.

Riganelli, O., Micucci, D., Mariani, L., 2018. Increasing the reusability of enforcers with lifecycle events. In: Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. ISOLA.

Riganelli, O., Micucci, D., Mariani, L., 2019. Controlling interactions with libraries in android apps through runtime enforcement. ACM Trans. Auton. Adapt. Syst. 14 (2), 8:1–8:29.

Riganelli, O., Micucci, D., Mariani, L., Falcone, Y., 2017b. Verifying policy enforcers. In: Proceedings of the International Conference on Runtime Verification. RV.

Romdhana, A., Merlo, A., Ceccato, M., Tonella, P., 2022. Deep reinforcement learning for black-box testing of android apps. ACM Trans. Softw. Eng. Methodol. 31 (4).

Sabnani, K., Dahbura, A., 1988. A protocol test generation procedure. Comput. Netw. ISDN Syst. 15 (4), 285–297.

Siami Namin, A., Andrews, J.H., Murdoch, D.J., 2008. Sufficient mutation operators for measuring test effectiveness. In: Proceedings of the 30th International Conference on Software Engineering. pp. 351–360.

Sidhu, D.P., Leung, T.-K., 1989. Formal methods for protocol testing: A detailed study. IEEE Trans. Softw. Eng. 15 (4), 413–426.

Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., Keromytis, A.D., 2009. ASSURE: Automatic software self-healing using rescue points. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS.

Taylor, P., 2023. Forecast number of mobile users worldwide 2020–2025. https://www.statista.com/statistics/218984/number-of-global-mobile-users-since-2010/.

Utting, M., Pretschner, A., Legeard, B., 2012. A taxonomy of model-based testing approaches. Softw. Test. Verif. Reliab. 22 (5), 297–312.

XDA, 2023. Xposed. http://repo.xposed.info/.

Oliviero Riganelli is an Associate Professor at the University of Milano-Bicocca. With a background in software engineering, he is actively engaged in cutting-edge research to enhance the reliability, safety and resilience of software systems. His passion lies in advancing autonomous computing, software testing and analysis. Dedicated to developing innovative methodologies and technologies, he aims to strengthen the autonomy and resilience of software, ensuring robustness in the face of dynamic challenges. His commitment to advancing software engineering is reflected in the numerous papers he has authored, which have appeared in prestigious journals and conferences such as TSC, TAAS, ICSE, and ISSTA.

Daniela Micucci obtained her Ph.D. in Mathematics, Statistics, Computational Sciences and Computer Science from the University of Milano in 2004. She is currently an associate professor at the University of Milano - Bicocca. Her main research interests are in the software engineering field, with a particular focus on software architectures. She is responsible for the Software Architecture Laboratory at the University of Milano - Bicocca. Daniela Micucci has published more than 70 papers at conferences and journals, including TSE, TAAS, JSS, ICST, and ICSE.

Leonardo Mariani received the Ph.D. degree in computer science from the University of Milano Bicocca in 2005, and he is currently full professor at the same university. His research interests include software engineering, in particular software testing, program analysis, automated debugging, and self-adaptive systems. He has authored more than 100 papers appeared at top software engineering conferences and journals. He has been awarded with the ERC Consolidator Grant in 2015, an ERC Proof of Concept Grant in 2018, and the ICST Most Influential paper 2023.