Anonymizing Test Data in Android: Does It Hurt?

Elena Masserini
e.masserini2@campus.unimib.it
University of Milano - Bicocca
Milan, Italy

Davide Ginelli
davide.ginelli@unimib.it
University of Milano - Bicocca
Milan, Italy

Daniela Micucci
daniela.micucci@unimib.it
University of Milano - Bicocca
Milan, Italy

Daniela Briola
daniela.briola@unimib.it
University of Milano - Bicocca
Milan, Italy

Leonardo Mariani
leonardo.mariani@unimib.it
University of Milano - Bicocca
Milan, Italy

ABSTRACT

Failure data collected from the field (e.g., failure traces, bug reports, and memory dumps) represent an invaluable source of information for developers who need to reproduce and analyze failures. Unfortunately, field data may include sensitive information and thus cannot be collected indiscriminately. Privacy-preserving techniques can address this problem anonymizing data and reducing the risk of disclosing personal information. However, collecting anonymized information may harm reproducibility, that is, the anonymized data may not allow the reproduction of a failure observed in the field. In this paper, we present an empirical investigation about the impact of privacy-preserving techniques on the reproducibility of failures. In particular, we study how five privacy-preserving techniques may impact reproducibilty for 19 bugs in 17 Android applications. Results provide insights on how to select and configure privacy-preserving techniques.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

data anonymization, privacy-preserving, privacy, bug reproduction, mobile applications, debugging, testing.

ACM Reference Format:
Elena Masserini, Davide Ginelli, Daniela Micucci, Daniela Briola, and Leonardo Mariani. 2024. Anonymizing Test Data in Android: Does It Hurt? . In *5th ACM/IEEE International Conference on Automation of Software Test (AST 2024) (AST '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3644032.3644463

1 INTRODUCTION

Collecting bug reports and information about the failures experienced by end-users while interacting with their applications is extremely important to reveal bugs [23, 24], and improve the quality and the reliability of the applications. Indeed, several problems are detected only once the software has been released [9], and the extensive collection of failure data is a key factor to enable the reproduction of the bugs, and later their correction.

Several approaches have been defined to reproduce failures from runtime data extracted from the field. For instance, failures have been reproduced starting from the flow of events executed in the app immediately before a crash [17], from executions traces with the operations performed before a crash [4, 10, 15], as well as from the content of the stack trace [16, 19], and bug reports [21, 22]. Despite the benefit of collecting data from the field to reproduce failures, user data can be fairly collected only by taking the sensitivity of the data into consideration. Indiscriminately collecting data may reveal sensitive information that should not be available outside the boundary of the app. For instance, failure traces may include sensitive information such as age, gender, financial data, and personal interests.

In specific cases, data can be partially anonymized. For instance, concerning the data stored in databases, *kb*-Anonymity can be used to mitigate the issue of sharing sensitive information through the databases used for testing [2]. When the execution path of the failure is available and symbolic execution is applicable to the target program, new synthetic executions that reproduce the failures might be derived to also mitigate issues with sensitive data [3, 11, 12].

The field of data mining has been investigating this challenge for several years defining a number of privacy-preserving techniques that can be used to alleviate the problem of incidentally disclosing sensitive information [8, 14, 20]. These techniques work by applying generalization or suppression operations to the data, so that the original information is not immediately available anymore [13]. For instance, a string 123456 representing an account number could be automatically rewritten as a random string of the same length, such as XHFPRT. These techniques can be readily applied to the data collected from the field to prevent disclosing sensitive data to third-parties.

While privacy-preserving techniques can clearly eliminate, or reduce, privacy issues, their impact on the capability of revealing failures has not been studied so far. Indeed, using anonymized data to reproduce failures is harder than using clear text data. That is, *protecting the privacy of the users and facilitating the reproduction of failures experienced in the field are two competing goals.*

In this paper we propose the first, to the best of our knowledge, empirical study about the impact of privacy-preserving techniques

This work licensed under Creative Commons Attribution International 4.0 License.

AST 2024, April 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0588-5/24/04.
https://doi.org/10.1145/3644032.3644463

on failure reproduction. We focused our study on failures experienced by users of mobile apps due to the popularity of mobile applications and their exposure to privacy issues [7]. Our study considers 19 bugs in 17 open source Android applications, and discloses insights about the trade-off between guaranteeing the privacy of the users and easing the reproduction of failures. In particular, we show that there is no unique choice about the privacy-preserving techniques to be used. Different contexts may require different techniques depending on the aspect to privilege.

This paper is organized as follows. Section 2 introduces and rigorously defines privacy-preserving techniques. Section 3 describes the design of the experiment conducted to evaluate the impact of privacy-preserving techniques on failure reproduction. Section 4 reports the empirical results and answers our research questions. Section 5 discusses related work. Finally, Section 6 provides final remarks.

2 PRIVACY-PRESERVING TECHNIQUES

Privacy-preserving techniques can be used to effectively anonymize data. This section defines the techniques that we considered in our study, describing how we adapted them to the problem of failure reproduction, when necessary.

Privacy-preserving techniques are typically used in the context of data mining, especially with records of databases. In fact, data contained in tables do not usually satisfy privacy requirements, and thus they cannot be shared without applying anonymization operations [8]. These operations may target individual records or sets of records. The former class of operations is useful when a third-party that accesses the data can essentially access only to individual records, and cannot compare the (anonymized) records between them. The latter class of operations is useful when a third-party can access the full set of records, and thus may infer facts by cross-analyzing the content of multiple (anonymized) records. In such a case, privacy-preserving techniques must consider the full set of records when anonymizing the individual records to prevent the incidental disclosure of sensitive information.

In the case of software failures, they are usually experienced once a while for the released apps, and only few failures are normally collected from a same user. To guarantee that user data is fairly collected, the application of strategies specifically designed to deal with large sets of repeated failures collected from the same users is likely not needed. For this reason, we focus on privacy-preserving techniques that can be applied to individual records.

In this paper, we consider failure traces consisting of streams of GUI events executed on the app when the failure occurred, as done in many failure reproduction techniques, such as CaRCrash [17] and ReCDroid [22]. That is, a failure trace is a sequence of events (a_i, w_i, d_i) where a_i is a GUI action (e.g., a click event) performed on a widget w_i (e.g., a button) possibly using data d_i (e.g., the text entered into an input field). The set of values d_i that occur in a failure trace are the data values subject to the anonymization process. We do not consider in our experiment the anonymization of other elements, such as the action or the widget. That is, we study how to prevent the failure trace from disclosing information such as the age, the address, or the personal income through the data values d_i entered into a form, while it is out of the scope of the study to hide the fact that a user has registered into an app by clicking on the REGISTER button.

The operations performed by privacy-preserving techniques to anonymize data vary based on the type of data. In particular, it is possible to distinguish three different classes of data to be analyzed: continuous values, categorical values, and string values. Continuous values are numeric values (e.g., someone's age or income) that can be used, for instance, as part of arithmetic operations. Categorical values are enumerated values that cannot be normally used as part of arithmetic operations [6]. Finally, string values are sequences of alpha-numeric characters.

We now present the privacy-preserving techniques based on the type of anonymization strategy they implement: generalization, suppression, and perturbation. Table 1 shows the specific techniques that we considered (Column *Technique*), classified according to the strategy they implement (Column *Strategy*) and associated with type of data that they can be applied to (Columns *Continuous*, *Categorical*, and *String*). The set of selected techniques reflects the taxonomy proposed by Mendes et al. [13]. We have excluded the *anatomization strategy* presents in the taxonomy, since it is strictly related to databases and cannot be applied to our context, and both the Top/Bottom Coding and the Post-Randomization (PRAM) techniques since our dataset does not include cases where they can be applied.

Table 1: Overview of privacy-preserving techniques.

Strategy	Technique	Continuous	Categorical	String
Generalization	Global Recoding	✓	✓	-
	Top Coding	✓	-	-
	Bottom Coding	✓	-	-
	Rounding	✓	✓	-
Suppression	Local Suppression	✓	✓	✓
	Special Char Driven LS	-	-	✓
Perturbation	Noise Addition	✓	-	-
	PRAM	-	✓	-

2.1 Generalization Techniques

Techniques that belong to this strategy replace values with more general ones [13]. These privacy-preserving techniques disclose some general information, while hiding the original value.

Global Recoding.

Definition: Global Recoding anonymizes a value by only disclosing information about the interval it belongs to. This technique can be applied to both categorical and continuous variables. In the former case, Global Recoding anonymizes a value by combining several categories into fewer ones. For example, if the categorical value represents different age groups (e.g., newborns, infants, toddlers, kids, and adults), Global Recoding may reduce them into two groups (e.g., 'baby' and 'kids or older'). In the latter case, Global Recoding replaces a variable with its interval. For example, the numerical age value can be replaced with its categorical age group [18].

Failure Reproduction: In the context of failure reproduction, replacing a categorical or continuous value with its interval (e.g.,

replacing the categorical input `infant` or the numerical input 1 with the category `baby`) would make the failure trace non-executable. In fact, the new value would not be processable by the application that expects either values from a specific enumeration of categories or numerical values. To obtain a processable input, and thus to attempt to reproduce the failure from the anonymized trace, the values anonymized with Global Recoding are then replaced with random concrete values within the anonymized interval.

Example: If a variable in the range [0, 10] is anonymized according to the sub-intervals [0, 5) and [5, 10], and the value to anonymize is 4.0, Global Recoding replaces the original value with the interval [0, 5). Failure reproduction shall generate random values within this interval to attempt to reproduce the failure. Similarly, if a categorical value `newborns` is anonymized with a more general category `baby` (that includes `newborns`, `infants`, and `toddlers`), test generation shall use values in the set `newborns`, `infants`, and `toddlers` to reproduce the failure.

Rounding.

Definition: This technique identifies several rounding points in the domain and maps the input value to be anonymized to the closest rounding point [6]. These rounding points could be identified by dividing the domain into multiple intervals, then selecting the middle point of each interval as rounding point.

Failure Reproduction: The anonymized value is an actual domain value and thus failure reproduction simply uses the value readily available in the trace.

Example: Given an input in the range (0, 10], the rounding points can be defined as the middle points of the intervals (0, 5) and [5, 10], that is, the values 2.5 and 7.5. Every value to be anonymized is mapped to one of these two values.

2.2 Suppression Techniques

Techniques that belong to this strategy entirely drop the values to be anonymized, or retain minimal information, to protect privacy [13].

Local Suppression.

Definition: This technique can be trivially applied to any data type (continuous, categorical, and string), since it replaces the input value with a missing value, whose semantics depends on the context [18]. For example, considering a record in a database, the corresponding missing value is `NULL`. In the context of Android applications, Local Suppression simply logs the empty string for any input value.

Failure Reproduction: In this case, failure reproduction is left with no information about the original value and thus it can only generate a random value coherent with the domain of the original value. In particular, if the original value is continuous, the technique generates a random value within the allowed range. If the original value is categorical, the technique chooses a random element from the set of possible values. If the original value is a string, the technique generates a string that matches a specific regular expression (in such a case, we consider both the case the new string has a length unrelated to the original string or has a length matching the original string).

Example: In all the cases, the anonymized value is the empty value. The generation is driven by the full range of values allowed by the input field. For instance, a random number between 0 and 100 could be generated for an input field representing the age of a person.

Special Char Driven Local Suppression.

Definition: Since sometimes bugs are triggered by anomalous characters that cannot be parsed or processed correctly, we defined a version of the Local Suppression that only preserves the special characters (defined as any non-alphanumeric character, such as *, !, and ?) contained in the value to anonymize. Special characters usually reveal virtually nothing about the original input, but they might be helpful to reproduce misbehaviors.

Failure Reproduction: The generation works the same than in Local Suppression, but the special characters in the input value are copied in random places within the generated value.

Example: Given the value *example!* to be anonymized, the technique generates a new random string that includes the special character `!`, such as `HQb!Ha`.

2.3 Perturbation Techniques

Techniques that belong to this strategy replace the original values with synthetic values close to the original ones [8, 13].

Noise Addition.

Definition: This technique is typically applied to continuous variables (i.e., to numbers). The general idea is to change the original value by adding or multiplying a stochastic or randomized number (i.e., the noise) to the original data [18]. Given a domain range $[min, max]$, a percentage of noise amplitude n, and a value to anonymize v, the technique generates a random value in the interval $[v - n * (v - min), v + n * (max - v)]$.

Failure Reproduction: The anonymized value is an actual domain value and thus failure reproduction simply uses the value readily available in the trace.

Example: Given a range [0, 10], a noise 0.30, and the value to anonymize 8.0, Noise Addition generates a random value in the interval $[8.0 - 0.30 * (8.0 - 0), 8.0 + 0.30 * (10.0 - 8.0)] = [5.6, 8.6]$.

3 EXPERIMENT DESIGN

3.1 Goals and Research Questions

The goal of this study is to investigate the impact of privacy-preserving techniques on the capability to reproduce the failures experienced in the field. To this end, we framed the following research questions.

***RQ1 - Effectiveness*: What is the failure-reproduction rate for anonymized failure traces?** This research question studies how failure-reproducing test cases derived from failure traces

Table 2: Reproducible Android application faults

Application	Domain	Version	Fault description
Binary Eye	Barcode scanner	1.56.2	Generating QR codes from some string values cause the app to crash.
Birday	Birthdays and events	1.9.0	The app crashes if the user adds a birthday on February 29th.
Catima Loyalty	Card management	2.16.0	Although only the initial of a card name should be shown in the icon, certain combination of initial characters are all erroneously shown in the icon.
Catima Loyalty	Card management	2.8.0	Expiry date for cards set before 1970 are not shown.
Contact Diary	Event and contact tracker	1.2.0	The app crashes when the input includes a malformed event duration that does not match the pattern hh:mm, such as `:mm, hh:`.
Debitum	Debts and lents tracker	1.4.0	Transactions are sometimes saved with an amount that slightly differs from the entered one.
Did I take my meds?	Medicine tracker	1.6.2	Some combinations of system time and edited time cause the edited time to be saved as P.M. even if it was A.M., and vice versa.
EinkBro	Browser	8.21.0	Some queries are considered as URLs and lead to webpage not found error.
Food Scale Droid	Grocery management	1.2	App crashes when weights contain a comma.
GrowTracker	Gardening	2.5.1	Entering a value with a dot in *from date* or *to date* fields causes the app crash.
Money Wallet	Accounting	4.0.4.1	Initial amounts in wallets can be incorrectly saved.
NoNonsense Notes	Notes	5.5.1	Closing and re-opening the app after a SD synchronization causes the loss of all notes contained in lists whose name includes a / .
Simple Calendar	Calendar	6.19.0	When importing birthdays from Simple Contacts, the age shown for contacts born before 1970 is incorrect.
Simple Money Tracker	Accounting	0.8.9	If the transaction amount is too big, the app crashes.
SplitBills	Shared expenses	0.3.10	Group names containing a / as a middle or final symbol are not correctly exported.
Tasks	Habit tracker	9.7.3	When creating a task, subtask names containing the sequence `' @'` are truncated.
To don't	Negative habits tracker	1.1	App crashes - changes are not saved when editing a task with an apostrophe in the name.
Track & Graph	Personal data tracker	1.5.1	Using a | symbol in the name of the first option in multiple option values causes the app to crash.
Track & Graph	Personal data tracker	1.5.1	A wrong number is saved if the used decimal separator does not match the one defined in the Android settings.

are impacted by privacy-preserving techniques. That is, it investigates how hard reproducing failures is, if the source traces are anonymized with the techniques presented in Section 2.

***RQ2 - Cost*: How many runs are necessary to reproduce failures with high confidence?** This research question studies the number of test executions that must be performed to establish if either the failure has been reproduced or the failure cannot be reproduced from the anonymized trace.

***RQ3 - Information Disclosure*: How often is the original value disclosed?** This research question investigates how often the anonymized value is reconstructed while reproducing a failure, thus potentially revealing sensitive information that should remain hidden.

3.2 Selection of the Subject Android Apps and Faults

To select the subject apps and faults, we performed an extensive manual analysis to look for failures that depend on user data, that is, failures that can be observed only if certain input values are entered. We restricted our selection to open-source F-Droid [5] apps with repositories present in either GitHub or GitLab, to make sure it is possible to inspect the app and actually identify the fault responsible for a given failure. We considered apps in the Money, Science & Education, Sports & Health, Time, Internet, and Writing categories, since these apps have a better chance of exploiting user inputs than apps in categories like Theming and Connectivity.

For every category, we manually checked at least 50 apps per category to identify the ones that have fillable fields, considering both the screenshots and the descriptions on their own page in F-Droid. For every identified app, we checked all the issues labeled as "bug" (or simply all the issues when labels are not available) to select the issues that are reported to be caused by specific inputs. We identified a total of 29 potentially useful issues spanning 26 apps.

To verify the presence of these issues, we downloaded the APK file corresponding to the version with the issue and reproduced the failure as reported in the issue. When the APK was not available, we checked out the correct version from the GitLab or GitHub repository of the application and generated the compiled app ourselves with Android Studio. We also checked the code of the app to determine if two same failures of a same app were originated by a same fault. We then classified failures as *reproducible* or *non-reproducible*, depending on the possibility to reproduce the failure with either an automatic Espresso [1] test case or a failure reproducing routine. In particular, we consider a failure non-reproducible if we could neither reproduce it with Espresso nor we could establish a clear relationship between the inputs and the fault present in the app.

We found eight non-reproducible failures and two identical failures generated by faults that were already included in the selection. We ended up with 19 reproduced input-dependent failures caused by distinct bugs in 17 Android apps. Detailed information about all the considered cases is publicly available in our repository, alongside with the material needed to reproduce the experiments and the results that we obtained: https://gitlab.com/sal-unimib-anonymization/experimentation. Table 2 reports the apps, their domain and version, and a description of the bugs present in the apps.

For each reproduced failure, with the exception of two cases where the app was incompatible with Espresso, we recorded an automatic Espresso test case that exposes it. For the two cases of

incompatibility, we inspected the faulty code in the apps and implemented a failure reproducing routine that given an anonymized input determines if the fault is exposed.

3.3 Configuration of the Privacy-Preserving Techniques

Depending on the nature of the data that must be anonymized, the techniques presented in Section 2 may require to be properly configured. In the following, we describe the configurations that we used.

String values can be anonymized with the Local Suppression and the Special Char Driven Local Suppression techniques. In both cases, the new value that must replace the original one is obtained according to a regular expression. We use the following four regular expressions that capture the cases we encountered in our subject apps: `[!-~]`, when all possible string values including special characters are allowed; `[A-Za-z0-9]`, when only alphanumeric values are allowed; `[0-9.,]`, when only numbers with any decimal separator are allowed; and `[0-9,]` or `[0-9.]` or `[0-9:]`, when only number with specific separators are allowed. All these cases are summarized in table Table 3.

Table 3: Regular expressions for string values.

Value type	Regex
All possible string values with special char.	`[!-~]`
Alphanumeric values	`[A-Za-z0-9]`
Numbers allowing both the decimal separators	`[0-9.,]`
Numbers allowing only a specific separator	`[0-9,]` or `[0-9.]` or `[0-9:]`

When the Special Char Driven Local Suppression technique is used and the original string includes one or more special characters, these special characters are inserted in random places within the new anonymized string. We configure the length of the generated string in two ways, experiencing both in our evaluation. That is, the length of the generated string can be random or equal to the length of the original string. In the case of random length, we use the interval [1-25] for short inputs (e.g., a note title or a loyalty card name) and [1-150] for long inputs (e.g., a description). In case the length of an input is bounded to a value lower than the maximum defined by these intervals, we set the maximum length to the maximum length accepted by the text field.

Numeric values can be anonymized with most of the privacy-preserving techniques. Local Suppression anonymizes values by generating new values within a specified interval. If the value to anonymize has boundaries defined by the application (e.g., the time can be only assigned with a value in the interval [0-24]), we configure the technique with these limits. Otherwise, we set the interval depending on the nature of the value: when a small value is expected (e.g., an age), we use the interval [0-100], otherwise if bigger values can be used (e.g., a currency) we use the interval [0-1.000.000]. Global Recoding and Rounding require the definition of the number of partitions to be used to split the interval of definition. Consistently with the previous definition of small and big values, we run the techniques with three configurations (using 2, 3, and 4 partitions) when a small value is expected by the app, and we use three different configurations (using 50, 100, and 500 partitions) when a big value is expected. Finally, we experience three different noise values (30%, 40%, and 50%) for Noise Addition.

The configurations that we used for the techniques applicable to numeric values are summarized in Table 4.

Table 4: Configurations for techniques applied to numbers.

Technique	Parameters	Possible Config.
All	Interval of Definition	1) As in the app 2) [0-100] 3) [0-1.000.000]
Global Recoding & Rounding	Number of Partitions	1) Small intervals: 2, 3, and 4 2) Big intervals: 50, 100, and 500
Noise Addition	Width of Noise	1) 30%, 40%, 50%

3.4 Experimental Procedure

To answer RQ1-3, we follow the procedure visually illustrated in Figure 1. We start from the Espresso test case that reproduces the bug as reported from the field by the user of the application, including the data reported in the original online issue. We identify ourselves a value coherent with the description in the issue, in the few cases a specific value was not available. To study the impact of privacy-preserving techniques, we anonymize the user data that the failure is dependent on with every applicable technique configured as discussed in Section 3.3.

The anonymization of the data resulted in a new Espresso test case that uses the values derived from the anonymization process rather than the original values. We then executed the new test and checked if the same failure could be reproduced after the anonymization process. Since anonymization and failure reproduction imply randomness, we repeat the anonymization process 100 times for every configuration, for a total of more than 11K test executions. All tests were executed on a Huawei P9 Lite smartphone with Android 9, except for the few cases that required a specific Android version and were tested on virtual devices. In the two cases of apps incompatible with Espresso, we executed our failure reproducing routine.

The implementation of the privacy-preserving techniques presented in this paper and the tool to run the failure reproduction process are publicly available in the following repository: https://gitlab.com/sal-unimib-anonymization/anonymization-android-tool.

The set of applications used in the study, the input that has been anonymized, the technique used for the anonymization, and the configurations used for the anonymization process are reported in detail in table Table 5. We add the labels Lo, Me, Hi next to the configurations present in the table, to identify the configurations

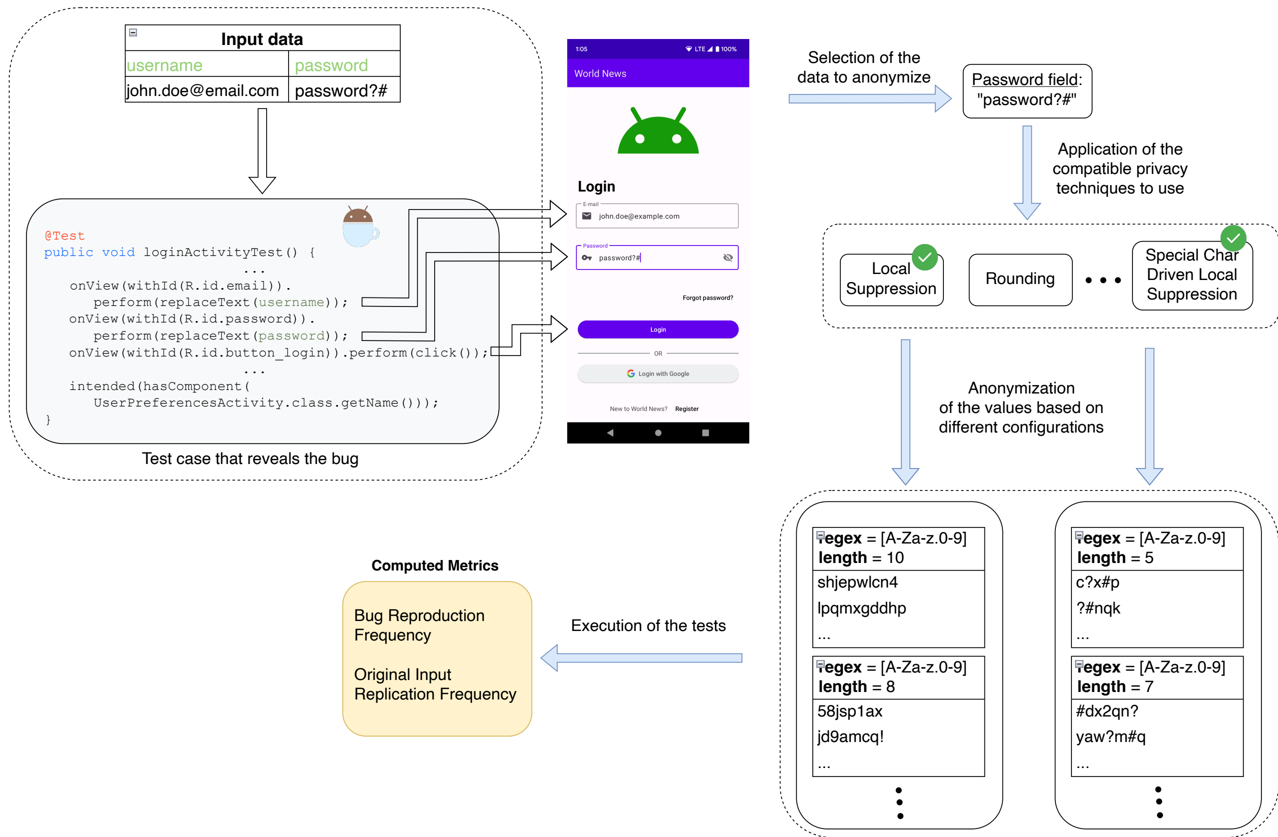

Figure 1: Overview of the experiment.

that retain less (Lo), medium (Me), or more (Hi) information from the original non-anonymized value.

To answer RQ1, we measure the *bug reproduction frequency*, that is, the ratio between the number of anonymized tests that reveal the same failure revealed by the original test and the number of anonymized tests. The more often a failure is revealed, the less impact a privacy-preserving technique has on the failure reproduction capability of the test cases. To answer RQ2, we compute the *number of repetitions* to reveal the original failure with a probability of 95%, that is, we estimate the number of tests that must be derived and executed from failure traces to be reasonably sure that a bug has been either reproduced or it is not feasible to reproduce it. To answer RQ3, we measure the *replication frequency of the original input*, that is, we measure the number of times the original non-anonymized value is generated during the failure reproduction process.

4 RESULTS

4.1 RQ1 - Effectiveness

Table 6 reports the bug reproduction frequency for the privacy-preserving techniques applied to strings and numbers.

Concerning the anonymization of strings, Local Suppression severely compromises the capability to reproduce failures (the mean success rate varies between 11% and 19%), depending on the configuration. The low bug reproduction frequency for Local Suppression is expected, since almost no information is retained from the original string. Interestingly, retaining more information from the original input (configuration Hi) has, in the majority of the cases, a negligible or negative effect on the bug reproduction frequency. This happens because preserving the length of the original string is often not a relevant factor in failure reproduction, while using (longer) random strings may increase the chance of using the right combination of characters that may trigger the failure.

In line with this intuition, SCD Local Suppression has a significantly better performance than Local Suppression (mean success rate of 49%). This confirms our intuition that by just disclosing a syntactic information that is largely irrelevant on the point of view of the user (i.e., the presence of a special character), failure reproduction might be often improved. Again, retaining the length of the string has not an impact on failure reproduction. Clearly, the presence of special characters alone is not always enough to reproduce failures. In these cases Local Suppression and SCD Local Suppression have similar performance, as for the Catima Loyalty and the Binary Eye apps. In some other cases, they are helpful but not sufficient alone, since the special character(s) might have to occur at a specific position, as in the first bug of the Track & Graph app, or in the context of a specific string, as in the Task app. In some

Table 5: Configurations of the techniques for each application's bugs analyzed.

App bug	Input	Technique	Configuration
Binary Eye	com.taobao.arthas.boot .ProcessUtils.findJavaHome (ProcessUtils.java:222)	Local Suppression & SCD Local Suppression	**regex**: [!- ~], **length**: equal to original (Hi) or [1-150] (Lo)
Birday	29 2 1996	Local Suppression	**interval**: [1-31] [1-12] [1937-2036]
		Global Recoding & Rounding	**interval**: [1-31] [1-12] [1937-2036], **partitions**: 2 (Lo) or 3 (Me) or 4 (Hi)
		Noise Addition	**interval**: [1-31] [1-12] [1937-2036], **noise width**: 0.3 (Hi) or 0.4 (Me) or 0.5 (Lo)
Catima Loyalty - bug 1	Atelier	Local Suppression & SCD Local Suppression	**regex**: [!- ~], **length**: equal to original (Hi) or [1-25] (Lo) **regex**: [A-Za-z0-9], **length**: equal to original (Hi) or [1-25] (Lo)
Catima Loyalty - bug 2	19 4 1963	Local Suppression	**interval**: [1-31] [1-12] [1900-2100]
		Global Recoding & Rounding	**interval**: [1-31] [1-12] [1900-2100], **partitions**: 2 (Lo) or 3 (Me) or 4 (Hi)
		Noise Addition	**interval**: [1-31] [1-12] [1900-2100], **noise width**: 0.3 (Hi) or 0.4 (Me) or 0.5 (Lo)
Contact Diary	:30	Local Suppression & SCD Local Suppression	**regex**: [0-9:], **length**: equal to original (Hi) or [1-5] (Lo)
Debitum	4.60	Local Suppression	**interval**: [0-100)
		Global Recoding & Rounding	**interval**: [0-100), **partitions**: 2 (Lo) or 3 (Me) or 4 (Hi)
		Noise Addition	**interval**: [0-100), **noise width**: 0.3 (Hi) or 0.4 (Me) or 0.5 (Lo)
Did I Take My Meds	15:30 20:36	Local Suppression	**interval**: [0-23] [0-59] [0-23] [0-59]
		Global Recoding & Rounding	**interval**: [0-23] [0-59], **partitions**: 2 (Lo) or 3 (Me) or 4 (Hi)
		Noise Addition	**interval**: [0-23] [0-59], **noise width**: 0.3 (Hi) or 0.4 (Me) or 0.5 (Lo)
EinkBro	how to open design.psd	Local Suppression & SCD Local Suppression	**regex**: [!- ~], **length**: equal to original (Hi) or [1-25] (Lo)
Food Scale Droid	543,	Local Suppression & SCD Local Suppression	**regex**: [0-9,.], **length**: equal to original (Hi) or [1-25] (Lo)
Grow Tracker	3.6	Local Suppression & SCD Local Suppression	**regex**: [0-9,.], **length**: equal to original (Hi) or [1-25] (Lo)
		Local Suppression	**interval**: [0-100)
		Global Recoding & Rounding	**interval**: [0-100), **partitions**: 2 (Lo) or 3 (Me) or 4 (Hi)
		Noise Addition	**interval**: [0-100), **noise width**: 0.3 (Hi) or 0.4 (Me) or 0.5 (Lo)
Money Wallet	4362.65	Local Suppression	**interval**: [0-1000000)
		Global Recoding & Rounding	**interval**: [0-1000000), **partitions**: 50 (Lo) or 100 (Me) or 500 (Hi)
		Noise Addition	**interval**: [0-1000000), **noise width**: 0.3 (Hi) or 0.4 (Me) or 0.5 (Lo)
NoNonsense Notes	list/name	Local Suppression & SCD Local Suppression	**regex**: [0-9,.], **length**: equal to original (Hi) or [1-25] (Lo)
Simple Calendar	1 1 1960	Local Suppression	**interval**: [1-31] [1-12] [1900-2100]
		Global Recoding & Rounding	**interval**: [1-31] [1-12] [1900-2100], **partitions**: 2 (Lo) or 3 (Me) or 4 (Hi)
		Noise Addition	**interval**: [1-31] [1-12] [1900-2100], **noise width**: 0.3 (Hi) or 0.4 (Me) or 0.5 (Lo)
Simple Money Tracker	20000000000000000000	Local Suppression & SCD Local Suppression	**regex**: [0-9.], **length**: equal to original (Hi) or [1-25] (Lo)
SplitBills	group/name	Local Suppression & SCD Local Suppression	**regex**: [!- ~], **length**: equal to original (Hi) or [1-25] (Lo)
Tasks	Subtask 1 @home	Local Suppression & SCD Local Suppression	**regex**: [!- ~], **length**: equal to original (Hi) or [1-25] (Lo)
To Don't	task'name add	Local Suppression & SCD Local Suppression	**regex**: [!- ~], **length**: equal to original (Hi) or [1-25] (Lo)
Track & Graph - bug 1	option|	Local Suppression & SCD Local Suppression	**regex**: [!- ~], **length**: equal to original (Hi) or [1-25] (Lo)
Track & Graph - bug 2	2.7	Local Suppression & SCD Local Suppression	**regex**: [!- ~], **length**: equal to original (Hi) or [1-25] (Lo)
		Local Suppression	**interval**: [0-100)
		Global Recoding & Rounding	**interval**: [0-100), **partitions**: 2 (Lo) or 3 (Me) or 4 (Hi)
		Noise Addition	**interval**: [0-100), **noise width**: 0.3 (Hi) or 0.4 (Me) or 0.5 (Lo)

Table 6: Bug Reproduction Frequency

	Strings					Numbers									
	Local Sup		SCD Local Sup			Local Sup	Global Recoding			Rounding			Noise Addition		
	Lo	Hi	Lo	Hi			Lo	Me	Hi	Lo	Me	Hi	Lo	Me	Hi
Binary Eye	0%	0%	0%	0%	Birday	0%	0%	2%	1%	0%	0%	0%	6%	5%	8%
Catima Loyalty - bug1	8%	10.5%	9%	9.5%	Catima Loyalty - bug 2	39%	62%	100%	39%	100%	100%	0%	56%	64%	60%
Contact Diary	8%	16%	49%	38%	Debitum	5%	9%	5%	6%	0%	0%	0%	6%	5%	6%
EinkBro	0%	1%	3%	7%	Did I Take My Meds	52%	100%	49%	100%	100%	0%	100%	74%	84%	90%
Food Scale Droid	61%	28%	92%	90%	Grow Tracker	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Grow Tracker	77%	27%	100%	100%	Money Wallet	4%	5%	9%	13%	0%	0%	0%	8%	8%	8%
NoNonsense Notes	18%	8%	100%	100%	Simple Calendar	27%	74%	100%	36%	100%	100%	0%	63%	63%	61%
Simple Money Tracker	1%	16%	5%	15%	Track & Graph - bug 2	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
SplitBills	10%	9%	81%	93%											
Tasks	0%	0%	20%	6%											
To Don't	18%	9%	99%	99%											
Track & Graph - bug 1	3%	2%	13%	21%											
Track & Graph - bug 2	35%	15%	59%	53%											
Mean	**18%**	**11%**	**49%**	**49%**		**41%**	**56%**	**58%**	**49%**	**63%**	**50%**	**38%**	**52%**	**54%**	**54%**

'Lo', 'Me' and 'Hi' in the header refer to configurations that retain less, medium, or more information from the input

other cases the fault was dependent on the semantic of the value and the mere presence of the special character was not enough to reproduce the failure.

For numeric inputs, Local Suppression is the technique with the lowest success rate (41%), with only Rounding - Hi performing worst (38%). Noise Addition and Global Recoding perform similarly: the effectiveness of Global Recoding ranged between 49% and 58%, and Noise Addition ranged between 52% and 54%. Rounding performed best in some cases, but with higher performance variance, with an effectiveness between 38% and 63%. Global Recoding and Rounding are more sensitive than Noise Addition to the choice of the configuration. In fact, Noise Addition works with intervals that are defined around the original input value. On the contrary, the intervals used in Global Recoding and Rounding are independent from the original input value, which could fall very near or on the edge of the interval, affecting the reproduction probability in cases where the fault is caused by values near to the original one.

The characteristics of the failure to be reproduced also have an impact. In fact, there are some easy cases where most of the techniques were systematically successful, for instance due to values formatted according to the Android settings that systematically generate failures if incompatible with the expectation of the app. We also had some hard cases with a failure reproduction rate below 10%. This is due to the small set of domain values that trigger the failure (e.g., in Birday only February 29 of leap years lead to the malfunction, over all the possible dates). In cases where the bug was caused by values in a range close to the original input (e.g., Simple Calendar, Did I Take My Meds, Catima Loyalty), Local Suppression is affected by its inability to preserve any information, leading to an overall lower success rate compared to other techniques. Noticeable, although if with low probability, it was always possible to replicate a bug using Global Recoding and Noise Addition, while Rounding tends to quickly reveal the failure or miss it.

For three apps, the attempt to reproduce the original failure led to the discovery of new bugs. In Binary Eye some strings used to generate a QR code differ from the ones obtained when scanning the code generated by the app (e.g., `}c:+8ha` when coded and then decoded becomes `~c:+8ha`). In To Don't some task names when saved cause all the other task names to be cancelled and replaced by a null value (e.g., the random string `S 0O}_('` was sufficient to reveal the bug). In Track & Graph - bug 1, the bug makes the app crash while creating a new multiple-value tracked habit, when the first option ends with '|', but we also discovered that option names with '||' cause the habit to not be saved.

Answer to RQ1: SCD Local Suppression should be preferred to Local Suppression when applied to strings, since it significantly increases the failure reproduction capability, while disclosing minimal information (the presence of a special character in the original string). Local Suppression applied to numbers had a significant, but not dramatic, impact on failure reproduction (41% success rate). Techniques approximating and perturbing the original value might increase the success rate (up to 63% in our experiments) at the cost of disclosing some information about the original value. Deciding how much information preserving from the input values should be done carefully, since preserving information not correlated with the failure trigger may negatively influence reproduction.

4.2 RQ2 - Cost

To measure the cost of using anonymized data to reproduce failures instead of using actual values, we computed the number of attempts (i.e., generation of anonymized values and then generation of the concrete test cases) that must be completed to reproduce the original failure with a probability of 95%. Table 7 shows the Mean and Max number of attempts necessary across all faults for a given technique and configuration, the number of non-reproduced faults (row # NR), and the results per fault. Note that a higher average success rate does not imply fewer attempts in average since there is a logarithmic relationship between reproduction probabilities and number of attempts.

When anonymizing strings, Local Suppression introduces a cost hardly affordable in practice, with close to 60 test generations and executions attempts needed in average, and up to 299 attempts in

Table 7: Iterations for Reproducing Failures with 95% probability

Strings

	Local Suppression		SCD Local Suppression	
	Lo	Hi	Lo	Hi
Binary Eye	-	-	-	-
Catima Loyalty - bug 1	36	28	32	31
Contact Diary	36	18	5	7
Einkbro	-	299	99	42
Food Scale Droid	4	10	2	2
Grow Tracker	3	10	1	1
NoNonsense Notes	16	36	1	1
Simple Money Tracker	299	18	59	19
SplitBills	29	32	2	2
Tasks	-	-	14	49
To Don't	16	32	1	1
Track & Graph - bug 1	99	149	22	13
Track & Graph - bug 2	7	19	4	4
Mean*	**55**	**60**	**21**	**15**
Max	**299**	**299**	**99**	**49**
# NR	**3**	**2**	**1**	**1**

Numbers

	Local Sup	Global Recoding			Rounding			Noise Addition		
		Lo	Me	Hi	Lo	Me	Hi	Lo	Me	Hi
Birday	-	-	149	299	-	-	-	49	59	36
Catima Loyalty - bug 2	7	4	1	7	1	1	-	4	3	4
Debitum	59	32	59	49	-	-	-	49	59	49
Did I Take My Meds	5	1	5	1	1	-	1	3	2	2
Grow Tracker	1	1	1	1	1	1	1	1	1	1
Money Wallet	74	59	32	22	-	-	-	36	36	36
Simple Calendar	10	3	1	7	1	1	-	4	4	4
Track & Graph - bug 2	1	1	1	1	1	1	1	1	1	1
Mean*	**23**	**15**	**32**	**49**	**1**	**1**	**1**	**19**	**21**	**17**
Max	**74**	**59**	**149**	**299**	**1**	**1**	**1**	**49**	**59**	**49**
# NR	**1**	**1**	**0**	**0**	**3**	**4**	**5**	**0**	**0**	**0**

'Lo', 'Me' and 'Hi' in the header refer to configurations that retain less, medium, or more information from the input
**mean value is calculated only over defined values (- cases are ignored in the computation of the mean)*

the worst case. SCD Local Suppression is more practical since it requires between 15 and 21 attempts in average, with a maximum between 49 and 99.

When working with numbers, Noise Addition seems to be the most affordable solution, since it reproduced all failures with 95% confidence with a mean number of attempts between 17 and 21 and up to 59 in the worst case (Me). Global Recoding is a good choice too, as it performs similar to Noise Addition except with Birday, which determines the higher mean and max values for this technique. This difference is due to the necessity of preserving day and month unchanged in the original input date (29-02-1996) to reproduce the failure, which is more likely to happen with Noise Addition since it creates intervals around the input value. Rounding can be useful to reduce the failure reproduction effort, but it also severally reduces the number of reproduced failures. Local Suppression is a valid alternative to Noise Addition when no information about the original value has to be disclosed, at the risk of failing to reproduce some failures.

Answer to RQ2: SCD Local Suppression is a cost-effective solution to anonymize strings. Numbers can be feasibly addressed with Noise Addition or Local Suppression, depending on the amount of information that can be disclosed.

4.3 RQ3 - Information Disclosure

We computed how often the failure reproduction process has generated the value before the anonymization during failure reproduction. Table 8 shows the percentage of cases it happened for the various combinations of apps and techniques. In case of strings, obtaining the original value is unlikely to happen due to the size of the space of possibilities. In fact, it happened only once for one app, where the format of the string was particularly constrained by the regular expression. In case of numbers, the replication of the original value happened slightly more frequently, with Noise Addition being responsible of the highest number of cases (which anyway consists of only three cases with a probability below 1.67%). This is due to the relatively smaller size of the numeric domain and the type of perturbations introduced by Noise Addition. Interestingly, it was not strictly necessary to reconstruct the original value in any of these cases, so the reproduction of the value was incidental and the user would not be really aware of this fact. The only exception is Birday where the failure requires the date `29-2-year` where `year` is any leap year. Thus the user would discover the date and month of the birthday, and would restrict the birthday to leap years.

Table 8: Frequency of replication of the original value.

App	Technique	Replication Freq.
	String	
Track & Graph - 2	Local Suppression	0.50%
	Numbers	
Birday	Global Recoding	0.67%
Birday	Noise Addition	1.67%
Catima Loyalty - 2	Noise Addition	0.67%
Debitum	Noise Addition	0.33%

Answer to RQ3: All the anonymization techniques largely hide the values they are applied to. Sometime the failure reproduction process may generate the original input in the attempt to reproduce the failure. This is unlikely to happen frequently, with only Noise Addition causing the reproduction of the original input in some rare cases.

4.4 Threats to Validity

A threat is about the limited set of bugs considered in the experiments. To mitigate this threat, we systematically searched for real bugs contained in open source applications and we selected Android applications from different categories to have multiple contexts in which to experiment with privacy-preserving techniques. The construction of the experimental dataset that we publicly released is

already the result of significant manual effort with hundreds of apps and reports manually inspected, as described in Section 3.2. Enlarging this dataset to address new domains is part of our future work.

Another concern is about the way we configured the privacy-preserving techniques. To avoid introducing any bias, we defined a configuration policy that we described in the paper. All the configurations are finally reported in our online repository.

Finally, another concern is related to the correctness of the implementation of the privacy-preserving techniques that we used for the experiments. To mitigate this threat, we extensively tested our tools and made our artifacts publicly available.

5 RELATED WORK

The studies most related to our work concern with the approaches designed for the anonymization of the data collected during failures, and with solutions for bugs reproduction.

One of the first approaches designed for releasing private data in the context of testing and debugging activities, while ensuring people's privacy, is *kb*-Anonymity [2]. This approach exploits symbolic execution and *k*-anonymity to generate anonymized database tuples that do not alter the behavior of the program, that is, the same program path is executed when the program uses both the original and anonymized values. The approach is limited to numbers and programs whose code is accessible and analyzable with symbolic execution. Castro et al. [3] investigated a similar approach but applied to the data included in crash reports. MultiPathPrivacy [11] and RESPA [12] investigated how to weaken the requirement about preserving the same execution path when introducing anonymized values by identifying alternative paths that shall still lead to the reproduction of the same bug.

Different from this body of work, we studied the effectiveness of privacy-preserving techniques that have been extensively applied to databases and that can be easily used to anonymize data collected during failures, without running any complicated analysis on the code of the application. The results reported in this paper provide useful insights about their effectiveness and cost, and the specific configurations that best fit the problem of anonymizing failure data.

Our work also relates to failure reproduction. We target the case of reproducing failures from (anonymized) failure traces collected from Android applications. The reproduction of failures from similar *non-anonymized* traces has been also considered in other works, such as CaRCrash [17] that collects and dispatches failures traces every time a failure is detected. Similarly, CrashDroid [19] can reconstruct replayable scripts from stack traces collected during failures.

Some other techniques considered reproducing failures from bug reports using NLP techniques, such as S2RMiner [21] and ReCDroid [22]. In this study, we considered the impact of anonymization techniques on failure traces and the corresponding test cases. We left to future work investigating more in details the impact of privacy-preserving techniques on test cases derived from bug reports, although in principle the artefact used to derive the test cases should not significantly affect the conclusions of our study.

Finally, some techniques addressed the problem of reproducing failures in Java, such as, BugRedux [10], JCHARMING [15], STAR [4], and EvoCrash [16]. We targeted Android since apps are often used to process personal information. Investigating other technical contexts is part of our future work.

6 CONCLUSIONS

Analyzing and reproducing failures from failure traces is important to timely fix faults and develop reliable applications. However, failure traces may disclose sensitive information about the users of the applications, and must be properly anonymized before they can be used for failure reproduction. This paper studies how privacy-preserving techniques extensively exploited in the context of database systems can be adapted to the problem of failure reproduction, and presents an empirical evaluation that discloses findings about their effectiveness and cost. In particular, our results show that the SCD Local Suppression technique introduced in this paper can be effective with the anonymization of strings, while numbers can be effectively anonymized with Local Suppression or Noise Addition, depending on the possibility to disclose some information about the original value that was anonymized. Our future work concerns with experiencing and studying privacy-preserving techniques applied to additional domains, such as Web Applications.

ACKNOWLEDGMENTS

This work has been partially supported by the Engineered MachinE Learning-intensive IoT systems (EMELIOT) national research project (PRIN 2020 program Contract 2020W3A5FY).

REFERENCES

[1] Android Developers Official Website. 2021. *Espresso.* https://developer.android.com/training/testing/espresso Accessed: 2023-01-11.
[2] Aditya Budi, David Lo, Lingxiao Jiang, and Lucia Lucia. 2011. *kb*-anonymity: a model for anonymized behaviour-preserving test and debugging data. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/1993498.1993551
[3] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. 2008. Better bug reporting with better privacy. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* https://doi.org/10.1145/1346281.1346322
[4] Ning Chen and Sunghun Kim. 2015. STAR: Stack Trace Based Automatic Crash Reproduction via Symbolic Execution. *IEEE Trans. Software Eng.* 41, 2 (2015), 198–220. https://doi.org/10.1109/TSE.2014.2363469
[5] F-Droid Contributors. [n. d.]. *F-Droid: Free and Open Source Android App Repository.* https://f-droid.org/ Accessed: 2023-03-26.
[6] Josep Domingo-Ferrer. 2008. A Survey of Inference Control Methods for Privacy-Preserving Data Mining. In *Privacy-Preserving Data Mining - Models and Algorithms*, Charu C. Aggarwal and Philip S. Yu (Eds.). Advances in Database Systems, Vol. 34. Springer, 53–80. https://doi.org/10.1007/978-0-387-70992-5_3
[7] Fahimeh Ebrahimi, Miroslav Tushev, and Anas Mahmoud. 2021. Mobile app privacy in software engineering research: A systematic mapping study. *Information and Software Technology (IST)* 133 (2021), 106466. https://doi.org/10.1016/j.infsof.2020.106466
[8] Benjamin C. M. Fung, Ke Wang, Rui Chen, and Philip S. Yu. 2010. Privacy-preserving data publishing: A survey of recent developments. *ACM Comput. Surv.* 42, 4 (2010), 14:1–14:53. https://doi.org/10.1145/1749603.1749605
[9] Luca Gazzola, Leonardo Mariani, Fabrizio Pastore, and Mauro Pezze. 2012. An Exploratory Study of Field Failures. In *Proceeding of the IEEE International Symposium on Software Reliability Engineering (ISSRE).* https://doi.org/doi:10.1109/ISSRE.2017.10.
[10] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing field failures for in-house debugging. In *Proceeding of the International Conference on Software Engineering (ICSE).* https://doi.org/10.1109/ICSE.2012.6227168
[11] Pedro Louro, João Garcia, and Paolo Romano. 2012. MultiPathPrivacy: Enhanced Privacy in Fault Replication. In *Proceeding of the European Dependable Computing Conference (EDCC).* https://doi.org/10.1109/EDCC.2012.31
[12] João Matos, João Garcia, and Paolo Romano. 2015. Enhancing privacy protection in fault replication systems. In *Proceeding of the IEEE International Symposium*

on Software Reliability Engineering (ISSRE). https://doi.org/10.1109/ISSRE.2015.7381827

[13] Ricardo Mendes and João P. Vilela. 2017. Privacy-Preserving Data Mining: Methods, Metrics, and Applications. *IEEE Access* 5 (2017), 10562–10582. https://doi.org/10.1109/ACCESS.2017.2706947

[14] Suntherasvaran Murthy, Asmidar Abu Bakar, Fiza Abdul Rahim, and Ramona Ramli. 2019. A Comparative Study of Data Anonymization Techniques. In *Proceeding of the Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing (HPSC), and IEEE Intl Conference on Intelligent Data and Security (IDS)*. https://doi.org/10.1109/BigDataSecurity-HPSC-IDS.2019.00063

[15] Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Sofiène Tahar, and Alf Larsson. 2015. JCHARMING: A bug reproduction approach using crash traces and directed model checking. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. https://doi.org/10.1109/SANER.2015.7081820

[16] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2020. Search-Based Crash Reproduction and Its Impact on Debugging. *IEEE Trans. Software Eng.* 46, 12 (2020), 1294–1317. https://doi.org/10.1109/TSE.2018.2877664

[17] Junmei Sun, Kai Yan, Xuejiao Liu, Min Zhu, and Lei Xiao. 2019. Automatically Capturing and Reproducing Android Application Crashes. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion (QRS)*. https://doi.org/10.1109/QRS-C.2019.00106

[18] Matthias Templ. 2017. *Methods for Data Perturbation*. Springer International Publishing, Cham, 99–132. https://doi.org/10.1007/978-3-319-50272-4_4

[19] Martin White, Mario Linares Vásquez, Peter Johnson, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Generating reproducible and replayable bug reports from Android application crashes. In *Proceedings of the IEEE International Conference on Program Comprehension (ICPC)*. https://doi.org/10.1109/ICPC.2015.14

[20] Lei Xu, Chunxiao Jiang, Jian Wang, Jian Yuan, and Yong Ren. 2014. Information Security in Big Data: Privacy and Data Mining. *IEEE Access* 2 (2014), 1149–1176. https://doi.org/10.1109/ACCESS.2014.2362522

[21] Yu Zhao, Kye Miller, Tingting Yu, Wei Zheng, and Minchao Pu. 2019. Automatically Extracting Bug Reproducing Steps from Android Bug Reports. In *Proceedings of the International Conference on Software and Systems Reuse (ICSR)*. https://doi.org/10.1007/978-3-030-22888-0_8

[22] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G. J. Halfond. 2019. ReCDroid: automatically reproducing Android application crashes from bug reports. In *Proceedings of the International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2019.00030

[23] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schröter, and Cathrin Weiss. 2010. What Makes a Good Bug Report? *IEEE Trans. Software Eng.* 36, 5 (2010), 618–643. https://doi.org/10.1109/TSE.2010.63

[24] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. 2020. How Practitioners Perceive Automated Bug Report Management Techniques. *IEEE Trans. Software Eng.* 46, 8 (2020), 836–862. https://doi.org/10.1109/TSE.2018.2870414