Mitigating Anonymity Challenges in Automated Testing and Debugging Systems

Silviu Andrica and George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Modern software often provides automated testing and bug reporting facilities that enable developers to improve the software after release. Alas, this comes at the cost of user anonymity: reported execution traces may identify users. We present a way to mitigate this inherent tension between developer utility and user anonymity: automatically transform execution traces in a way that preserves their utility for testing and debugging while, at the same time, providing k-anonymity to users, i.e., a guarantee that the trace can at most identify the user as being part of a group of k indistinguishable users. We evaluate this approach in the context of an automated testing and bug reporting system for smartphone applications.

1 Introduction

To debug a software failure, one must understand its root cause; unfortunately this can be quite challenging, with many bugs taking weeks to diagnose [1]. Therefore, modern software often ships with built-in features for automatically collecting program execution information that enables developers to more quickly debug the software (e.g., Windows Error Reporting collected billions of traces that helped developers fix 5,000 bugs [2]).

Current error reporting systems sacrifice developer productivity to preserve user anonymity: they report some execution information (e.g., backtraces, some memory contents) but forgo other useful information, such as the data the program was processing and the execution path it was following when it crashed, due to user privacy and anonymity concerns. In this paper, we seek to strike a better balance between user anonymity and productivity-enhancing execution information.

We describe this technique in the context of ReMoTe, an automated testing and debugging system that helps programs to collaborate on doing some of the debugging work that developers do: ReMoTe records program executions, modifies them to generate tests, runs the tests, and validates discovered bugs. For each part of this process, ReMoTe provides a so-called *pod* (Figure 1); pods from different program instances collaborate via a *hive*.

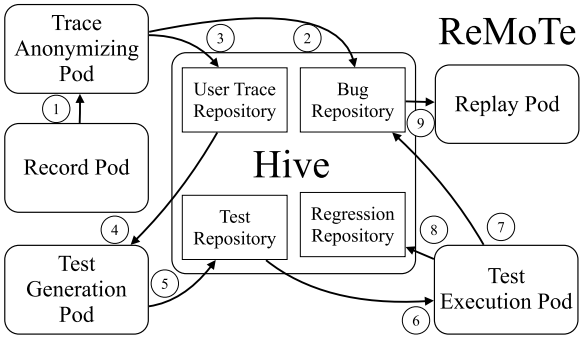


Figure 1: Overview of ReMoTe's architecture.

The *Record* pod logs a program's execution to an *interaction trace* that contains the program's interaction with the user and with the program's environment. Traces are added to local storage, but never removed.

ReMoTe is a distributed system: pods running on different machines can collaborate to generate tests, execute tests, and validate uncovered bugs by exchanging interaction traces. However, these traces contain information that may identify users. Thus, before a trace is shared with other pods, the *Trace Anonymizing* pod automatically transforms it to preserve the user's anonymity ①. Next, the pod informs the user about the amount of information the transformed trace contains that can identify her, and enables the user to veto the sharing.

There are two reasons why a ReMoTe user shares a trace with other pods: either the user experienced a bug, and the trace is put in the *Bug Repository* ②, or the user wishes to contribute the trace as a usage scenario, and the trace is put in the *User Trace Repository* ③. Developers can inspect the *Bug Repository* and use the *Replay* pod to replay bugs and understand how they occur ⑨.

The *Test Generation* pods modify traces from the *User Trace Repository* ④ to generate tests that populate the *Tests Repository* ⑤. *Test Execution* pods run these tests ⑥ and categorize them into bugs ⑦ and/or tests for the regression suite ⑧. These pods run inside the program that generated a trace or inside other program instances.

This paper presents the *Trace Anonymizing* pod. We describe means to quantify the degree of user anonymity

and the utility of a trace, and devise algorithms to improve anonymity without harming utility.

First, we quantify the anonymity of a user sharing a trace as the size k of the set of distinct users reporting an equivalent trace. Second, we define a trace to have debugging utility if replaying it triggers the root cause of the bug and the failure [3], and to have test-generation utility if it describes an execution generated by an actual user. Third, our system improves user anonymity by expunging personally identifiable information and ensuring the user behavior encoded in a trace is not unique to that user, while still maintaining replayability of the trace.

The contributions of this paper are: 1) two techniques that provide users with k-anonymity, one using dynamic program analysis, and another leveraging crowdsourcing; and 2) a new metric that quantifies the amount of personally identifiable information contained in a trace. We built a ReMoTe prototype for Android applications and showed that ReMoTe protects users' anonymity ($k >$ 100) and is more efficient than similar techniques.

In the rest of the paper, we define k-anonymity (§2), describe the two algorithms (§3–§4), evaluate our prototype (§5), review related work (§6), and conclude (§7).

2 Anonymity of Interaction Traces

This section defines interaction traces, describes the concept of k-anonymity that underlies our work, and defines a metric to quantify the amount of user-identifying information contained in a trace.

2.1 Interaction Traces and Event Types

We define an interaction trace T as a sequence of events, $T =< e_1, \ldots, e_n >$. Each event e_i records one of four sources of "non-determinism" that influence a program's execution: 1) user interaction with the program's GUI, 2) network communication, 3) input read from the machine the program runs on, and 4) decisions made by the runtime system. Replaying a trace T should consistently drive the program along the same execution path.

An event plays one of two roles during replay: *proactive events* cause a feature of the program to execute (e.g., click on the "Send SMS" button), while *reactive events* provide the feature with the input it needs (e.g., the phone number and SMS text). Events of both types may contain information that identifies users. Table 1 shows events for each source of non-determinism for interactive Android applications, and maps them to a role.

ReMoTe targets interactive programs, which generate these events at a rate bounded by the speed with which users interact with programs. Thus, compared to recording solutions that target events at a lower software layer (e.g., [4]), which are generated more frequently, the *Record* pod is more scalable, since it runs less frequently.

	User	**Network**	**Device**	**Runtime**
Proactive	Tap, Tilt, Scroll, Drag, Press key	Receive push notification	Trigger geofence	Fire timer
Reactive	Textbox value, Selector value, Slider value	Server response	Date, GPS location, Camera & Mic output, Shared data, Device settings	Async tasks scheduling

Table 1: Trace events for Android applications, classified by covered non-determinism source and proactivity role.

We say a trace contains personally identifiable information (PII) if it can be used to determine a user's identity, either alone or when combined with other information that is linkable to a specific user [5].

2.2 K-Anonymity

A data set satisfies k-anonymity if and only if each set of values that can be used to identify the source of a data element appears at least k times in the set [6], i.e., the source of an element cannot be narrowed down to fewer than k candidates. We say that each element of a data set satisfying k-anonymity is k-anonymous.

In [6], the data set is represented as a table PT, and each row contains information about a single subject. Some table columns contain private information (e.g., received medication), others provide identification details about the subject (e.g., birth date and zip code), but none contain information that explicitly identifies the subject (e.g., the name of the patient). Thus, one may naïvely conclude that table PT is anonymous.

K-anonymity quantifies the possibility of linking entries from the PT table with external information to infer the identities of the sources of the data in the PT table.

Consider there exists a set QI of columns in PT, called a quasi-identifier, (e.g., $QI = \{birth\ date, zipcode\}, PT = QI \bigcup \{medication\}$) that also appears in a publicly available table PAT. If the PAT table contains additional columns that explicitly identify its sources (e.g., $PAT =$ *Voters list* $= \{name\} \bigcup QI$), then an attacker can use the quasi-identifier values to join the two tables and learn private information about a subject (e.g., the medication a person receives). The attack is similar to executing an SQL join operation on the PT and PAT tables that uses the quasi-identifier as the join condition.

This attack relies on the value of the quasi-identifier being unique for each subject in the *PT* and *PAT* tables. To achieve *k*-anonymity, one must modify the *PT* table to break this assumption [7]. This is not necessary if, in the *PT* table, each quasi-identifier value already appears *k* times. If not, one can suppress the entries that prevent achieving *k*-anonymity, or repeatedly use generalization strategies to make the values of the quasi-identifier less precise (e.g., replace the birth date with the year of birth) until *k*-anonymity is reached, or add new entries to the table to make it satisfy *k*-anonymity (not covered in [7]).

We seek to prevent ill-intentioned developers and program users from abusing interaction traces to learn the identity of the user whose program recorded a trace.

A trace identifies its source through reactive events, which may contain explicit PII (e.g., usernames), or through proactive events, which detail user behavior. We aim to provide users with *k*-anonymity, which in our case represents the guarantee that a trace identifies its source as the member of a set of *k* indistinguishable users.

We say an interaction trace is *k*-anonymous if it is *k*-proactive-anonymous and *k*-reactive-anonymous. A trace is *k*-reactive-anonymous if, for each reactive event in the trace, there exist at least *k* alternatives (§3). A trace is *k*-proactive-anonymous if at least *k* users observed it (§4). Thus, a *k*-anonymous trace contains behavior exhibited by *k* users, and there are *k* alternatives for each program input contained in a reactive event.

We now describe the differences between the original *k*-anonymity technique [6] and ours:

First, ReMoTe computes the maximal *k* it can achieve for a trace's anonymity, it does not enforce a particular *k*.

Second, ReMoTe cannot detect a minimum, complete quasi-identifier, as is assumed in [6], because the structure of a trace is unconstrained and its length is unbounded. ReMoTe takes a conservative approach, by choosing completeness over minimality, and defines the quasi-identifier to span all the events in a trace.

Third, the equivalent of the *PT* table is distributed among users. While the ReMoTe hive could store all the observed, non-anonymized traces, doing so posses the risk of an attacker subverting the hive, gaining access to the raw traces, and thus being able to identify users.

Finally, the pods share a trace with the hive only once it has achieved *k*-anonymity. *K*-anonymity increases, for example, when adding a newly recorded trace to the set causes existing ones to become *k*-proactive-anonymous.

2.3 Amount of Disclosed Information

We define the *k-disclosure* metric to quantify the amount of PII in a trace T. We start from two observations: First, its value should be inversely proportional to how *k*-anonymous an observed trace T is, because the higher the *k*, the less specific to a user the trace is. Second, the amount of PII contained in a trace is emergent: while each event in the trace may be encountered by multiple users, the order of events in the trace may be unique.

We define the value of the *k*-disclosure metric for an observed trace T, k-disclosure(T), to be the sum of the inverses of the values quantifying how *k*-anonymous is each of T's subsequences, $k(trace)$. That is, $k\text{-disclosure}(T) = \sum_{1 \le i \le j \le |T|} \frac{1}{k(T_{ij} = <e_i, \ldots, e_j>)}$.

We expect k-disclosure(T) to decrease over time because, once a program observes a trace, it is permanently added to local storage; as more users encounter T or its subsequences, the trace's *k*-anonymity increases.

3 Anonymity of Reactive Events

Reactive events contain program inputs that can directly identify users, such as usernames. A reactive event is useful for replaying a trace T if it causes the program to make the same decisions during replay as it did during recording [3]. If one can replace a reactive event e_R^{orig} with $k-1$ reactive events e_R^{sub} without affecting the trace's ability to replay the program execution, then we say the trace T is *k*-reactive-anonymous with respect to event e_R^{orig}. More simply, we say e_R^{orig} is *k*-anonymous.

Consider the example of an Android application for sending SMS messages. The user fills in the destination phone number (reactive event e_R) and the message body. When the user presses the "Send" button, the application converts the phone number to a `long`. Say that the user entered a number starting with a '+' character, and the program crashes, but any string that does not start with a digit can reproduce this crash—ReMoTe can replace e_R with *k* alternatives, where *k* is the number of such strings.

To compute the number *k* of alternatives for a reactive event e_R, ReMoTe must know how the program makes decisions based on the program input associated with e_R. ReMoTe uses concolic execution [8] to collect the conditions, called path constraints, corresponding to the executed branch statements that depend on reactive events. The technique uses "symbolic" variables that encode constraints on values, instead of concrete values.

Next, for each reactive event, ReMoTe uses a constraint solver to compute the solutions that satisfy the path constraints referring to it. The number of solutions determines how anonymous trace T is w.r.t. that event.

ReMoTe uses the following algorithm. It replays each event e_i in a trace T. When replaying a reactive event, the algorithm copies $e_i.input$ (the program input contained in e_i) to $e_i.concrete$, marks $e_i.input$ symbolic, and adds an entry for e_i in the map tracking path constraints (*PC*). When the program branches on a condition involving the symbolic variable $e_j.input$, and both branch targets may

be followed, the algorithm forces the program to take the target that $e_j.concrete$ satisfies. The algorithm uses static analysis to decide whether to add the path constraint corresponding to the taken branch to the *PC* map and maintain $e_j.input$ symbolic, or to replace it with $e_j.concrete$. When replay finishes, the algorithm computes the number of solutions for each reactive event e_R.

ReMoTe iteratively computes the number of solutions for a set of path constraints *PC* by generating a solution, adding its negation to *PC*, and asking the solver for another solution. This process is time consuming, so ReMoTe bounds the number of solutions, establishing a lower bound for how k-reactive-anonymous is a trace.

ReMoTe modifies the trace to replace each program input contained in a reactive event with one of its computed alternatives, thus removing the PII from the trace.

The algorithm is similar to the one described in [9]. The difference is our use of static analysis to make concolic execution more efficient by avoiding the concolic execution of runtime-system code. This code affects only the execution of the runtime system, not the execution of the program and, thus, needlessly slows down concolic execution. The static analysis examines the stack trace of a program when it branches on a symbolic variable, and checks if the branch is *in* the program's code or if its result is *used* by the program—only in these two cases is the associated path constraint added to the *PC* map.

A technical report [10] presents a detailed description of the algorithm, the benefits of using the generated alternatives, their drawbacks, and mitigation solutions.

4 Anonymity of Proactive Events

Proactive events reveal a program's usage, and this usage could uniquely identify the user. For example, an employee may access a company application's features that are only accessible to executive management, and then a feature only accessible to financial department employees. By analyzing the corresponding proactive events, one could infer that the user is the company's CFO.

A related example is one where the two features are accessed in different traces, and each trace contains the same sequence of events that acts as a quasi-identifier and allows identifying the user.

To prevent user behavior details contained in a trace from identifying users, ReMoTe ensures that the entire trace (including reactive events) is k-proactive-anonymous *before* it is passed to the hive. Therefore, every trace seen by the hive corresponds to the executions of $\geq k$ distinct users, and it does not contain behavior specific to any one of them, so the trace's source cannot be identified more narrowly than that set of $\geq k$ users.

To check if a trace T is k-proactive-anonymous, ReMoTe can just query every program instance whether it experienced execution trace T in the past, and tally up the results. Alas, providing trace T to other program instances could compromise the user's anonymity.

The challenge is to design an algorithm that counts how many users observed the trace T without explicitly revealing T to them. Our solution is to hide T among a set S of traces, ask program instances whether they observed any of the traces in S, and probabilistically compute the number k of instances that indeed observed T.

The algorithm runs as follows: Program instance A, run by user U who wishes to share the trace T, constructs the query set S. The set S contains the hashes of trace T and of other traces that act as noise. Next, instance A sends the set S to the ReMoTe hive, which forwards the set S to each program instance A_i run by users U_i.

After the *Record* pod records an interaction trace, it saves the hashes of the trace and of its sub-traces to a history set H. When receiving a query set S, each instance A_i replies positively if its history set H_i contains any of the hashes in the set S, i.e, if $H_i \bigcap S \neq \emptyset$.

The ReMoTe hive counts the number K of positive replies and sends it to instance A, which computes the probability that k of the K instances recorded T—this determines how k-proactive-anonymous trace T is.

This algorithm protects the anonymity of the U and U_i users, because instances A_i cannot learn T, and instance A cannot learn the trace that caused A_i to reply positively.

ReMoTe runs the same algorithm for each of trace T's sub-traces and computes the amount of personally identifiable information contained in trace T, i.e., k-disclosure(T). Finally, instance A reports the k and k-disclosure(T) values to the user U. If the user agrees, instance A shares the trace T with the ReMoTe hive.

There are four challenges associated with this algorithm. First, instance A may be tricked into revealing the trace T by a sequence of carefully crafted queries. Second, instance A needs to generate feasible traces as noise for the set S. Third, to compute the number k of instances that recorded T, instance A must approximate the likelihood of instances A_i recording each trace from the set S. Finally, instance A may be tricked into revealing T by an attacker compromising the result of the voting process. [10] details these challenges and presents their solutions.

5 Empirical Evaluation

We built a ReMoTe prototype that can be used by Android applications. In this section, we evaluate it on PocketCampus [11], a client-server Android application that we modified to use ReMoTe.

We evaluate the anonymity ReMoTe can provide for reactive events by quantifying how k-anonymous each field of a server response is after PocketCampus processes it. We focus on two functionalities provided by

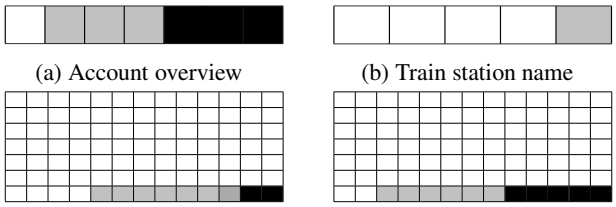
the application: check the balance of a student card account, and display the time of the next departure from a public transport station. In both cases, PocketCampus makes a request to a server, processes the response, and displays some information. Each server response contains multiple fields, and we computed how many alternatives ReMoTe can generate for each field.

Figure 2 contains the results, and shows that PocketCampus places few constraints on server responses, enabling ReMoTe to provide high k-anonymity for users.

Figure 2a shows that the server's response to inquires about the account balance contains seven fields: one is not processed by PocketCampus (the white box), three for which ReMoTe generates more than 100 alternatives (the gray boxes), and three for which ReMoTe cannot generate alternatives (the black boxes), because they contain floating point values (not supported by our constraint solver) or because PocketCampus checks their value against constant values (e.g., the server's status).

Figure 2b shows the server's response when queried about the name of a public transport station.

Figures 2c and 2d show the same server response, but in different interaction traces. Figure 2c corresponds to PocketCampus showing the departure time, while Figure 2d corresponds to additionally displaying trip details, which causes PocketCampus to further process the response and place additional constraints.



(a) Account overview (b) Train station name

(c) Next departure time (d) Details of next departure

Figure 2: k-anonymity for server response fields. White boxes show unprocessed fields, gray boxes show fields with $k \geq 100$, black boxes show fields with $k = 1$.

We report the percentage of bits identical in the original server response fields and the ones ReMoTe generated, considering only those processed by PocketCampus. Figure 3 shows that the alternative responses reveal, on average, 72% of the original bits, thus being more similar to the original ones than randomly-generated ones.

We evaluate the speedup in the concolic execution completion time brought by using the static analysis described in Section 3. Figure 4 shows that, when using the analysis' results, PocketCampus finishes processing a server response within 10 minutes, as opposed to more than one hour when the analysis is not used.

The technical report [10] evaluates additional aspects of ReMoTe, such as how k-proactive-anonymous is a trace and what is its storage space overhead.

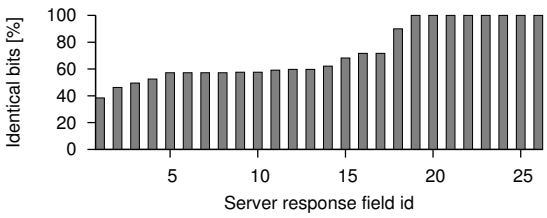


Figure 3: Percentage of bits identical in the original server responses and the alternatives ReMoTe generated.

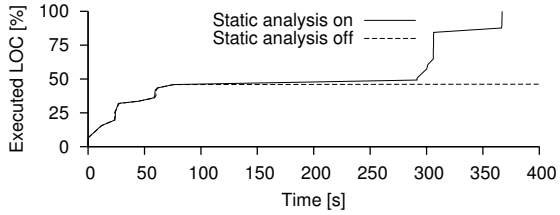


Figure 4: Concolic execution speedup obtained by using the static analysis described in Section 3. The figure can be read as a comparison, in terms of efficiency, between our algorithm and the one described in [9].

6 Related Work

Our work on generating alternatives for reactive events is most similar to [9], which also relies on collecting path constraints that describe decisions made by a program. The differences are that we use static analysis to discard path constraints that do not affect a program's execution, and we consider the anonymity threats related to a user interacting with a program. Camouflage [12] builds on [9] and introduces two techniques to enlarge the set of bug-triggering inputs, which ReMoTe can leverage.

Our crowdsourced k-anonymity technique is similar to the query restriction techniques pertaining to statistical databases [13], since one can view the crowd of program instances as a distributed database.

7 Conclusions

This paper describes two techniques to transform program execution traces to maximize users' anonymity, yet maintain the traces' utility for testing and debugging. One technique uses dynamic program analysis to expunge explicit personally identifiable information from traces, while the other leverages the crowd of users to verify that the user behavior encoded in a trace is not unique. We prototyped the techniques, and preliminary results suggest that they are effective and efficient.

References

[1] P. Godefroid and N. Nagappan, "Concurrency at Microsoft – An exploratory survey," in *Intl. Conf. on Computer Aided Verification*, 2008.

[2] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt, "Debugging in the (very) large: ten years of implementation and experience," in *Symp. on Operating Systems Principles*, 2009.

[3] C. Zamfir, G. Altekar, G. Candea, and I. Stoica, "Debug determinism: The sweet spot for replay-based debugging," in *Workshop on Hot Topics in Operating Systems*, 2011.

[4] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," in *Symp. on Operating Sys. Design and Implem.*, 2002.

[5] B. Krishnamurthy and C. E. Wills, "On the leakage of personally identifiable information via online social networks," in *Workshop on Online Social Networks*, 2009.

[6] L. Sweeney, "K-Anonymity: A model for protecting privacy," in *Intl. Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 2002.

[7] P. Samarati and L. Sweeney, "Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression," SRI International, Tech. Rep., 1998.

[8] K. Sen, "Concolic testing," in *Intl. Conf. on Automated Software Engineering*, 2007.

[9] M. Castro, M. Costa, and J.-P. Martin, "Better bug reporting with better privacy," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[10] S. Andrica and G. Candea, "Mitigating anonymity challenges in automated testing and debugging systems," Ecole Polytechnique Fédérale de Lausanne (EPFL), Tech. Rep. #186656, 2013.

[11] "PocketCampus," http://www.pocketcampus.org.

[12] J. Clause and A. Orso, "Camouflage: automated anonymization of field data," in *Intl. Conf. on Software Engineering*, 2011.

[13] R. Agrawal and R. Srikant, "Privacy-preserving data mining," in *ACM SIGMOD Conf.*, 2000.