MultiPathPrivacy: Enhanced Privacy in Fault Replication

Pedro Louro, João Garcia and Paolo Romano
INESC-ID Lisboa / IST
Lisboa, Portugal
{plouro,jog,romanop}@gsd.inesc-id.pt

***Abstract*—Most computer applications are published with bugs, whose reproducibility is strictly dependent on the availability of detailed information about the real usage of the application. Unfortunately, this data collection process raises severe privacy issues, as error reports are very likely to include personal information. This represents a strong disincentive for users to submit error reports, hampering the software maintenance process.**

In this work we address the issue of how to design data obfuscation mechanisms aimed at anonymizing the error reports generated by faulty applications, without compromising the bug reproducibility. The solution presented in this paper, MultiPathPrivacy, is based on an idea which is, to the best our knowledge, still unexplored in literature: maximizing the achievable degree of obfuscation by exploiting the presence of multiple execution paths leading to the manifestation of the same bug.

MultiPathPrivacy relies on an off-line reachability analysis phase, based on symbolic execution techniques, which is aimed at identifying not only the set of alternative execution paths leading to the execution of the code block where the bug manifested, but also to determine the symbolic constraints on the user inputs that are necessary to generate such execution paths.

By exploiting the presence of disjoint sets of alternative user inputs/execution paths leading to the manifestation of the same bug, MultiPathPrivacy allows achieving striking improvements of the anonymization quality when compared to state of the art solutions. Via an experimental study, based both on a real, privacy-sensitive application and on publicly available software repositories, we show that MultiPathPrivacy can achieve up to 87% reduction of the amount of user input information leaked by the error report, evaluated in terms of bits of information revealed, and percentage of residual non-anonymized input.

I. Introduction

Bugs have accompanied the development of software from its inception and are a challenge for programmers. Empirical observations suggest that the density of errors in industry code has remained relatively constant while the size (in terms of lines of code) of a software product has increased by several orders of magnitude. This means that the overall number of bugs is growing alarmingly [1]. In addition, errors can cause tremendous costs and even be fatal to human life [2].

Although more than half of the resources in a typical development cycle are invested in testing and bug fixing, bugs manifest themselves after software is released and persist long after [1]. A crucial factor in software errors is the limited knowledge programmers have about the behavior of the application when executed by its end-users and therefore the limitation in testing they face [3]. It is therefore of paramount importance to involve end users in the software maintenance process, since they have insights on how the application is used in reality. In fact, users do have interest in collaborating with the goal of improving their user experience on software that they already bought [4].

Unfortunately, existing tools for bug reporting are largely unsatisfactory both from the perspective of the end users and of the software maintenance team. Popular bug reporting tools, such as the well known Microsoft Windows Error Reporting (WER), which gathers information from millions of users around the world, in fact, transmit to the team maintenance information related to the system state at the time of failure. From the end user perspective, however, the distribution of such information raises important privacy issues, as explicitly admitted in the "Privacy Statement for the Microsoft Error Reporting Service [1]":

> *"For example, a report that contains a snapshot of memory might include your name, part of a document you were working on, or data that you recently submitted to a website. If you are concerned that a report might contain personal or confidential information, you should not send the report."*

From the programmer perspective, however, the information conveyed by the error report from WER may still be insufficient to allow for the deterministic reproduction of the failure, as the knowledge of the system's state at the moment of manifestation of the failure often does not provide sufficient clues about its root sources. In fact, the search for the causes of the state reported is one of the tasks in which the programmers spend more time [5].

What would greatly benefit the debugging process is indeed the availability of tools automating the deterministic replay of the bug at the software maintenance site. Achieving this result, however, would entail filtering out all sources of non-determinism in the application, by logging every event that can affect the reproducibility of the application run, transmitting it and replaying it transparently from the log during the debugging session.

Among all sources of non-determinism, user inputs (including accessed files and data exchanged via the network) are probably those associated with the higher risks of leakage of privacy sensitive information. These entries may in fact

[1] http://oca.microsoft.com/en/dcp20.asp

contain user information either of personal (e.g. name, address, credit card), professional (e.g. excerpts of documents), or even behavioural (e.g. references to Web sites or running applications) nature. In this paper we focus on this kind of privacy-sensitive sources of non-determinism, assuming the presence of orthogonal mechanisms to deal with additional sources of non-determinism (for instance, thread scheduling [6], [7]).

An interesting approach, which is at the core of several recently proposed solutions [8], [9] to this problem, is based on the idea of obfuscating input data "as much as possible", while still guaranteeing that it triggers the very same execution path that led to the manifestation of the bug. These solutions rely on symbolic execution techniques to derive automatically the so called "path conditions", namely a set of logical constraints on the user inputs sufficient to ensure the deterministic re-execution of the application. Data obfuscation is then performed by replacing the original user input with any random solution matching the path conditions clauses defined on the input.

These approaches have been proven to have good potential, even though their actual effectiveness depends on the structure of the execution path leading to the bug, and, more precisely, on the restrictiveness of the path conditions associated with it (namely, on the dimension of the solution space for the path conditions). In other words, as we will also confirm experimentally in the following, the data obfuscation quality achieved by these techniques can strongly vary depending on the structure of the application's code executed before actually triggering the bug.

The solution presented in this paper, MultiPathPrivacy (MPP), is based on an idea which is, to the best our knowledge, still unexplored in literature: maximizing the achievable degree of obfuscation by exploiting the presence of multiple execution paths leading to the manifestation of the same bug.

Analogously to the aforementioned data obfuscation solutions, also MPP relies on symbolic execution techniques to identify the set of logical constraints on the user inputs associated with code execution paths. Unlike, existing solutions, however, MPP relies on an off-line reachability analysis phase that is aimed at identifying, for each code block c in which a bug can manifest, the set of alternative execution paths leading to the execution of c, along with the symbolic constraints on the user inputs that are necessary to generate such execution paths. This allows the error reporting module, running on the client side, to verify, via automatic re-execution, which ones, among the set of alternative execution paths/user inputs, led to the reproduction of the user experienced bug.

By exploiting the presence of disjoint sets of alternative user inputs/execution paths, all leading to the manifestation of the same bug, MPP can generate obfuscated input data choosing from a (potentially) larger domain, defined by the disjunction of the path conditions associated with the different execution paths. This allows MPP to achieve striking improvements of the anonymization quality when compared to state of the art solutions for a twofold reason. On one hand, MPP can benefit from the existence of alternative execution paths imposing less restrictive path conditions (i.e. forcing to leak a minor amount of information on the original user input). On the other hand, MPP can enhance anonymization quality even if the individual path conditions associated with alternative execution paths (leading to the same bug) are not less restrictive than those associated with the original execution path. In fact, provided that these alternative path conditions are at least partially disjoint, their disjunction will yield a widening of the domain used to generate obfuscated data, and, therefore, an enhancement of the data obfuscation quality.

We quantify the enhancement of the obfuscation quality achieved by MPP by means of an experimental study, based both on a real, privacy-sensitive application, and on applications extracted from publicly available software repositories. Our experimental data show that, when compared to state of the art data obfuscation solutions analyzing exclusively the conditions of the original execution path, MPP can achieve up to 87% reduction of information leakage, evaluated in terms of bits of information revealed, and percentage of residual non-anonymized input.

The remainder of this paper is structured as follows. Section II discusses related work in data obfuscation for error reporting in software maintenance process. Section III overviews the MPP solution, discussing the key rationale underlying this proposal. The details of its architecture and implementation are provided in Section IV. The results of the experimental evaluation are reported in Section V. Finally, Section VI concludes the paper.

II. Related Work

Several approaches have been proposed with the primary objective of ensuring user privacy when sending error reports to reproduce failures. This section discusses the features and limitations of the most relevant ones.

A. Scrash

In order to protect the user's privacy, with regard to data that is sent in bug reports, Scrash [10] was the first proposed tool. The main goal of Scrash is to remove all sensitive information contained in the report just before it is sent to the maintenance team.

In the development phase of the application, all fields that contain sensitive information are marked as such. During the execution of the application, any field that relies on sensitive information is also marked as sensitive. Thus, marked sensitive data is propagated throughout the application. When a failure occurs and an error report is created, it is inspected and all sensitive information is removed. This is achieved since the sensitive data is grouped in a memory area bounded by special identifiers. As the error report contains the contents of memory at the time of failure, Scrash simply removes from the report any information that is included in memory areas marked by special identifiers.

This technique's main limitation is the fact that it requires the programmers to mark the sensitive data, as it is clearly

not reasonable for the user to do it. On the other hand, relying on programmers for this identification is equivalent to assume their trustworthiness. It is also necessary to have access to source code in order to make changes to it, which could be a great inconvenience to applications already on sale, as well as for those which do not have the code available. Additionally, this technique removes sensitive information from the report, which might mean a loss of relevant information that could potentially hamper, or even prevent, the successful reproduction of bugs.

B. Better Bug Report

Preventing the loss of relevant information to the failure reproduction is one of the objectives of Better Bug Report (BBR), a work proposed by Castro et al. [8]. This technique assumes that the application is being continuously monitored in order to log the input introduced by the user. When an error is detected the log is used to replay the execution, previously instrumented in order to gather the execution trace. The trace contains the sequence of instructions executed by each thread and the concrete values of source and destination operands of each instruction. The detailed execution trace is then used to perform a symbolic execution along the path followed by the failed execution to compute path conditions. The path conditions are the set of restrictions on the user input that forces the execution along one path. Any input that satisfies the same path conditions from the failed execution causes the software to follow the same execution path until it fails. BBR feed these path conditions to a Satisfiability Modulo Theories (SMT) solver to compute a new input that satisfies the path conditions, but is otherwise unrelated with the original input. The new input is included in the error report to allow software vendors to reproduce the bug. The report reveals only the information in the path conditions: all inputs that satisfy the path conditions generate the same error report. So the entropy loss can be measured with the *bits of information revealed* by the path conditions. This metric represents how much information is revealed by the technique by calculating how many inputs satisfy the path condition. The user privacy depends on the input domain that satisfies the path conditions. In some cases, even with the application of this technique, all the input is revealed. This happens when there is only one input solution that satisfies the path conditions that led to the bug.

C. Camouflage

A very similar technique was performed in the tool Camouflage [9], which adopts the same approach of BBR, although using different mechanisms for the generation of path conditions and their resolution in the form of a new input. In order to generate the path conditions that led to the failure, also Camouflage performs symbolic execution in the application. To this end, it makes use of an extension of Java PathFinder[2], a tool originally created for software model checking, which has been extended to be applied as well to the area of symbolic execution. In order to calculate the path conditions, Camouflage uses the overloading capabilities of Java PathFinder to modify the original application code, replicating its semantics but making the necessary steps to record the path conditions. The path conditions are then sent to YICES[3], the SMT solver used by Camouflage. For the assessment of the privacy loss, also Camouflage uses the metric of *bits of information revealed*, as Castro et al. This work also pointed out the shortcomings of this metric, highlighting how it fails to provide any measure of the actual portion of the disclosed user inputs. In order to address this shortcoming, Camouflage proposes another metric called *residue*. *Residue* represents the percentage of the original input that remains unchanged after obfuscation. Camouflage is only applicable to Java programs, while the technique of Castro et al. can be used on any application that runs directly on x86 family of processors.

```
a=readInput();
b=0; //error!!
if (a==1) {
  c=5;
} else {
  c=10;
}
//now, the fault!!
d=(a+c)/b;
```

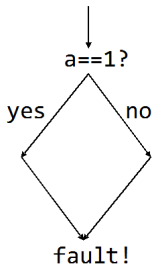

Figure 1. Code example for the exploitation of multiple execution paths.

III. Overview of the MultiPathPrivacy system

In general, in order for a bug to be replayed, the application needs to reach a given erroneous state. In the particular application execution in which the bug manifested, that erroneous state was reached at the execution of a specific instruction of the application (see for example the division by zero in Figure1). MPP exploits the key idea that the same bug can manifest itself at a given instruction of a program, even if that instruction is reached via different execution paths.

Figure 1 illustrates a simple example of a scenario in which the knowledge of the existence of alternative execution paths triggering the same bug can be exploited to achieve a significant enhancement of the data obfuscation quality achieved by the deterministic fault replication system. In fact, in approaches such as BBR or Camouflage, which consider exclusively the original execution path, the path conditions associated with the example in Figure 1, namely $(a = 1)$, would force to fully reveal the value of the user input. On the other hand, in that example, even if the user input that is assigned to variable a is not 1, and the `else` path is taken, the bug manifests itself anyway.

MPP takes into account the existence of multiple execution paths leading to the bug manifestation by deriving the path

[2]http://babelfish.arc.nasa.gov/trac/jpf

[3]http://yices.csl.sri.com/tool-paper.pdf

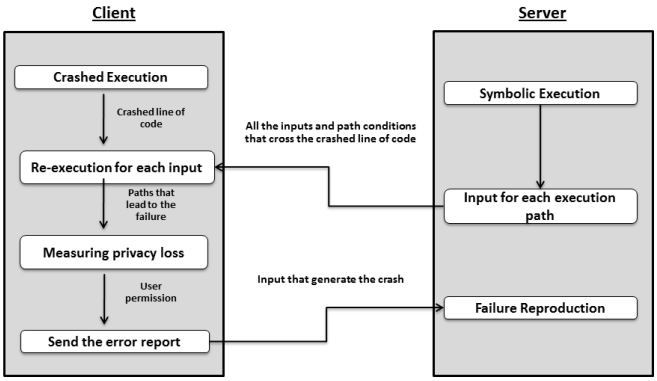

Figure 2. Overview of the MultiPathPrivacy architecture.

conditions used to obfuscate the user input from the disjunction of the path conditions of all these execution paths. This can (and normally does) lead to a relaxation of the original path conditions and to the achievement of higher obfuscation levels. In our example, being both $(a = 1)$ and $(a \neq 1)$ path conditions leading to the same application fault, their disjunction leads to a total relaxation of the constraint on a with an associated increase of user privacy (as the resulting error report can achieve full obfuscation of a's value).

IV. MPP Architecture

The MPP architecture (see Figure 2) contains two distinct components, namely the client component and server component, which run, respectively, on the user's and on the maintenance team's execution environment. The division of functionalities between the two components aims at offloading computation-intensive tasks to the maintenance team's infrastructure.

When the client of the error report module detects a crash in a given line of code, it inquires the server to obtain, for each different execution path that leads to the execution of that line of code, the set of logical constraints on the inputs that lead to the execution path, as well as an instance of such inputs.

Next the client replays automatically, typically in background, the set of execution paths using the inputs provided by the server component. This allows the client to determine which execution paths actually lead to the replay of the original fault, and to determine, as a disjunction (i.e. logical union) of the corresponding path conditions, the domain of alternative inputs that would lead to the fault reproduction.

Then, the obfuscated input is produced and displayed to the final user, along with compact statistics quantifying the amount of information leaked from the original input. The transmission of the error log to the server remains conditioned, as usual, to a final approval by the end user.

If this authorization is granted, the application developers will be able to automatically replay the same bug experienced by the user (namely, a bug occurring at the same line of code, and of the same nature) but using obfuscated input data possibly associated with (possibly very) different execution paths.

A. Server Component

In MPP the server component has the function of transferring some of the computational effort from the client to the maintenance side. The key role role of the server component (in addition to receiving the final error report from the client) is to externalize a service allowing clients to obtain information on the alternative execution paths leading to the execution of a given code block.

The implementation of the server component was performed taking into account four fundamental requirements for this component:

1) Search for alternative paths that traverse a given line of code.
2) Computation of the path conditions for those alternative executions.
3) Generation of alternative user inputs satisfying each of the alternative path conditions.
4) Deterministic replay of the execution received as an error report.

Regarding the first two requirements, it is necessary to make a path space traversal of a given application while storing the path conditions. These path conditions not only serve to uniquely identify a given path, but are later used to generate a input that traverses that path. In MPP, these requirements are satisfied by relying on symbolic execution [11], a program analysis technique that represents the inputs with symbolic values instead of concrete ones, and executes the application while manipulating the conditions that are being applied to the symbolic variables. To support the symbolic execution in MultiPathPrivacy, we used Symbolic Pathfinder (SPF) [12], a tool that executes Java bytecode symbolically through a customized interpretation that adds the symbolic logic in a totally transparent fashion. This tool was created as an extension of Java PathFinder [13], a model checker that is used in our case, to systematically generate and execute the symbolic tree of code under analysis.

Given that the symbolic execution of an entire application can become too heavy in terms of memory and time, SPF was built in order to be able to run the application in concrete mode (i.e. disabling symbolic execution), and switch to symbolic mode execution only inside relevant methods. SPF receives as input the application to examine and information about which methods should be executed symbolically. SPF also offers the choice to specify, which ones, among the parameters of a method, should be treated as symbolic or as concrete. At the end of its execution, SPF provides a summary of the symbolic execution of each method. This summary contains, among others, the execution paths represented by their conditions and the instance of a possible input that can trigger the execution of each one of these paths.

As already mentioned, In MPP we use the SPF symbolic execution engine to extract information on the set of logical constraints on user input (with the logical conditions expressed considering the input binary encoding, analogously to what is done in [8], [9]) leading to the execution of a given line of

code of the (part of) application under debugging. This implied altering SPF's *modus operandi* in several directions.

A first step towards the integration of SPF within the MPP system was to automatically identify the portions of Java applications that can collect user inputs during execution, and to ensure that SPF was instructed to treat the variables used to store user inputs as symbolic. In fact, originally SPF was conceived to receive, as its sole input, the input parameters of the method for which symbolic execution was requested. This result was achieved in a modular way, by exploiting a number of extension mechanisms offered by JPF (a detailed description of these mechanisms can be found in [14]), and intercepting a (relatively) small number of lower level methods in the Java library (e.g. the `nextInt()` from the `Java.Util.Scanner` library).

A more structural change to SPF was instead required to derive the conditions of the execution paths that pass through any line of code of the program. SPF was in fact originally designed to present just a summary of the different execution paths as a result of the symbolic analysis of individual methods. In order to achieve the desired result, we altered SPF's behavior to make it store, during the symbolic execution of a Java bytecode instruction, the corresponding line of code (identified uniquely by the class which it belongs to, and its line number) and associated path condition in an ad-hoc structure. At the end of the full symbolic analysis, this structure can be used to identify all the *distinct path* conditions associated with a line of code in the application. Furthermore, it is necessary to solve the path conditions on each of the symbolic variables that stored the original user inputs (that as we have mentioned will be used by the client applications during the replay phase). Given that the path conditions stored in the structure during symbolic analysis phase may be redundant, in order to maximize performance, we implemented also a cache for solved conditions. Thus, before solving a given condition, it is first tested whether it has been already solved and, if so, the solution is retrieved from the cache. As a result, the modified version of SPF that we integrated in the MPP system produces, for each line of code, the set of path conditions that cross it, and, for each path condition, an input instance exercising that execution path is also calculated.

Finally, in order to support the deterministic replay of the application fed with the obfuscated inputs eventually produced by the client module, we integrated in our solution the *replayer* module of LEAP [6], a recent framework for the deterministic replay of concurrency bug that has served also as the basis for the client component of MPP, as it will be detailed in Section IV-B.

B. Client Component

Upon the failure of an application (we here consider only fail-stop bugs, detectable and classifiable by intercepting uncaught Java exceptions), the client module gathers the corresponding exception type and line number (LN) from the application's local crash report. The LN is then sent to the server component that returns the set of distinct execution paths that include that LN, encoded by means of the corresponding conditions on the user inputs, as well as an instance of the inputs that trigger each of these execution paths. These inputs are individually tested through re-runs of the application. In each re-execution, it is checked whether the application incurs in the same kind of crash or not. If it does, the tested execution path reproduces the failure and, therefore, the domain of possible inputs that reproduce the failure is extended to include the corresponding path conditions. In fact, the full domain of inputs that produce the same failure can be computed as the disjunction of the path conditions of all the paths, among those returned by the server, that actually reproduce the failure. For example, in the case of Fig.1 when we expand the path conditions from $(a == 1)$ to $(a == 1) \vee (a \neq 1)$, we expand the domain of the user input variable a from the specific value 1 to all possible values of the variable. The level of privacy provided to the user is then calculated, by means of various metrics quantifying the amount of information about the original input that can be inferred from the error report.

The design of the client component was aimed at supporting two fundamental requirements:

1) Deterministic recording and replay
2) Measuring privacy loss.

Deterministic Recording and Replay. The starting point for the development of the deterministic recording and replay functionality in the MPP system has been LEAP [6], a recent system to support deterministic replay of concurrent programs written in Java. The main focus of LEAP is the deterministic replay of programs in multiprocessor environment and, to this end, LEAP records the order of shared memory accesses by different threads of execution.

The LEAP architecture is divided into three different components, namely the *transformer*, the *recorder* and the *replayer*. The input for the *transformer* is a Java application that results in two differently instrumented versions, one for the recording of relevant information during the execution, and another used during the bug replaying phase, called, respectively, *record* and *replay* versions. In short terms, the *transformer* chooses the points of the application code that must be manipulated to record/replay sources of non-determinism associated to concurrent memory accesses on shared memory elements.

LEAP, however, does not consider different sources of non-determinism, and, in particular, does not support deterministic recording/replaying of user inputs. This is clearly an essential feature for the MPP system, which was supported by having the LEAP *transformer* instrument the replay version so to intercept the calls to methods of the Java library related to input management, and to inject code for automatically feeding the user inputs returned by the server component.

Measuring Privacy Loss. From the set of path conditions corresponding to the executions that reproduce the failure, it is necessary to calculate the loss of privacy associated with

the disclosure of an error report. Note that, the error report contains only one randomly chosen failing input. However, the loss of privacy is calculated based on all the possible inputs that reproduce the failure.

A major advantage of the MPP approach is that, in addition to allowing for identifying alternative execution paths imposing less stringent restrictions on the domain of user inputs, it also allows for determining the cardinality of the set composed by all possible inputs leading to a manifestation of the bug.

Indeed, the number of distinct possible inputs that trigger the bug represents an accurate measure of user privacy as, the larger the domain of possible inputs is, the less sure an attacker on the MPP server side will be of whether the submitted input has any relation to the original input.

Previous systems [8], [9], [15] use the metric of bits of information revealed to quantify privacy. This is a measure of entropy, which quantifies uncertainty in terms of number of bits [16]. Intuitively, thi metric measures the amount of information that is revealed by calculating the number of different inputs that satisfy the restrictions on the input in order to replay a failure. Specifically, suppose that a variable A can assume, with equal probability, m different values. The entropy of A is calculated as $\log_2 m$. If the client reveals that the possible values of A are constrained at a proportion p of m, the information disclosed is measured in bits as:

$$\log_2 m - \log_2 pm = -\log_2 p$$

An obfuscated input I' reveals $\sum_{i \in I'} |\log_2(x_i)|$ bits of information about the original input I, where x_i is the number of solutions for the conditions that involve the variable i divided by the size of its input domain. For example, assuming that i_0 is an 8-bit character (i.e., its input domain contains 256 values) and that 100 of the 256 possible values satisfy the constraints on i_0. In this case, i_0 reveals approximately $\sum_{i \in I'} |\log_2(100/256)| = 1,36$ bits of the 8 total bits of information about i_0.

The bits of information revealed metric provides a good starting point for assessing the strength of the user input obfuscation mechanism. However, its results can be misleading. For example, it is possible to decrease the amount of bits revealed while leaving large portions of the input non-obfuscated. To illustrate this situation, consider a program that reads 10 characters as input. Assume that the constraints on each of the last 5 characters have 10 possible solutions, while the first 5 characters must remain the same (i.e. cannot be obfuscated). If the number of possible solutions for the last 5 characters is increased from 10 to 200, the amount of information revealed decreases from 63,3 bits to 41,7 bits. This decrease correctly indicates that it is now more difficult to recover the original input, but it fails to indicate that it was necessary to fully disclose half of the input, which that may be important, especially if the first half of the input happens to disclose more sensitive information than the second half.

In order to address this shortcoming, Clause et al. proposed a new metric to measure privacy loss, which they called *residue*. *Residue* is essentially the percentage of input that remains unchanged after obfuscation. For the example mentioned in the last paragraph, the percentage of *residue* would not change if the number of possible solutions for the last 5 characters increased from 10 to 200, indicating that obfuscation may not be as effective as *bits of information revealed* metric would suggests.

These metrics were used by various input anonymization techniques having in common their application to one path condition. In MultiPathPrivacy, which takes advantage of multiple execution paths to increase the user's privacy, it is necessary to adapt the evaluation to multiple path conditions. Intuitively, it is easy to see that if the input can be calculated from a larger domain (derived from the greater diversity of paths), it will be more difficult to infer the original input. In practice, this is done through the logical disjunction of the conditions of each of the paths that reproduce the failure. In particular, in the metric of *bits of information revealed*, the logical disjunction of the path conditions means that, for each user input i, the number of values, x_i, leading to the manifestation of the bug, is calculated taking into account the different, unique input values satisfying the conditions associated with each execution path, i.e. counting only once any input value that may satisfy the conditions associated with more that one execution path. These two metrics were applied to MultiPathPrivacy, and their results presented and analyzed in section V.

V. Evaluation

MultiPathPrivacy has been evaluated experimentally and compared to existing anonymization techniques. It is important to refer that the aim of MultiPathPrivacy is to increase user privacy in fault replication, through the analysis of multiple paths that reproduce the failure. The analysis of multiple paths is performed by the development team using symbolic execution before the application is deployed, while the monitoring and testing of execution paths is done in the user environment. Given these characteristics, the chosen evaluation criteria were:

- **Privacy Loss**: Quantifies the level of privacy that the customer takes when sending an error report. It is primarily with this criteria that we will compare the performace of MultiPathPrivacy with techniques which do not take advantage of the multiple paths that reproduce the failure.
- **Performance Overhead**: Observation of performance cost imposed by the implementation of MultiPathPrivacy in the client's environment both in the application monitoring and in the path testing.
- **Scalability**: The symbolic execution technique, applied in the maintenance team's environment to calculate alternative paths, has known scalability limitations which we will analise in the case of MultiPathPrivacy.

All experiments were performed on a AMD Phenom II x2 555 at 3.2 GHz machine, with 4 GB of RAM and Windows 7 64-bit operating system.

A. Subjects

To evaluate MultiPathPrivacy, we used a set of eleven case tests from three Java applications. The first two applications were made for [17] and are available at the *Software-Artifact Infrastructure Repository* [18]. The faults required for evaluation purposes were injected into the applications by creating mutant versions of these applications with MuJava [19] which change, for example, relational and conditional operators that may lead to failures in applications. The chosen applications were:

- **Cruise Control**: a program that simulates the engine and cruise control speed of a car. The failure in this application is triggered when it is requested to resume cruise control without it having been started before.
- **Elevator**: a program for controlling a set of elevators that serve a given number of floors. The failure in this application is triggered when a user selects a floor stop at a floor number that is greater than the total floor count.

To complete the assessment carried out with this test set, was also used a real application, iDate[4]. iDate is an application for mobile devices that aims to find people with a given profile chosen by the user criteria, for example, age, sex or height. iDate was adapted to run locally on a desktop computer. The failure exercised in this application is related to an implementation change between two different versions of the iDate. The representation of boolean values in the message that is exchanged between users. This representation is no longer done through a pre-defined text (eg yes or no), but with text that can be processed to boolean values (i.e. `true`, `false`). So when two users communicate, with different iDate version, there is a failure when processing the information.

Application	# Classes	LOC	Control Flow Instructions
Cruise Control	4	358	33
Elevator	8	581	56
iDate	3	1225	112

Table I
DESCRIPTION OF THE SUBJECTS USED TO EVALUATE MULTIPATHPRIVACY.

Table I characterizes each test application's source code. It is important to highlight that, although the Cruise Control and Elevator applications do not have an input that can be considered sensitive, they have a complex state machine in terms of number of states and transitions by events that become relevant to this assessment. This characteristic can be verified by the percentage of control flow instructions which are the main source of branching in program execution and consequently the main source of different states in the state machine representation of the program. iDate was chosen because it is a real world application of real use and handles extremely sensitive data.

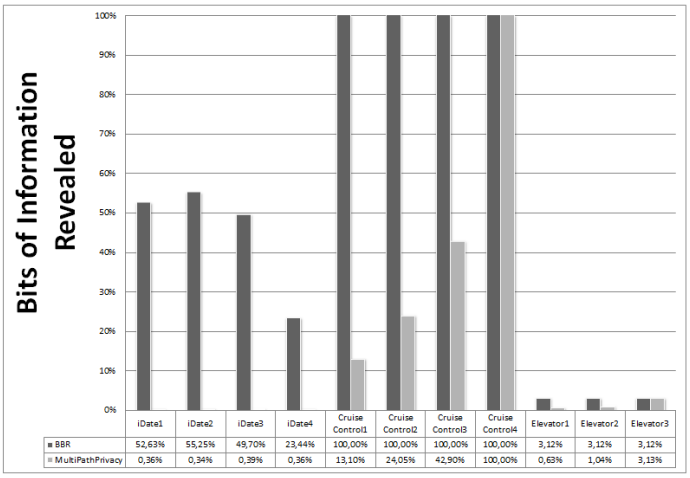

Figure 3. Results of *bits of information revealed* percentage in the test cases with MPP and Better Bug Reporting

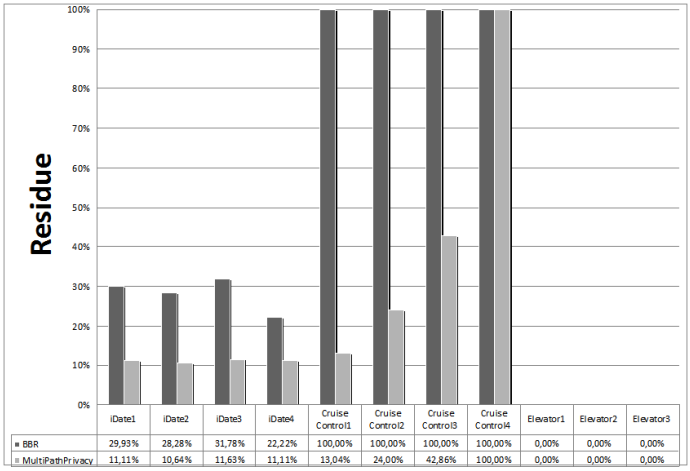

Figure 4. Results of *residue* percentage in the test cases with MPP and Better Bug Reporting

B. Results

This section presents the results of applying MultiPathPrivacy to the test cases presented in the previous section, measured by the metric of *bits of information revealed* and *residue* as explained in section IV-B. Its results will be compared to a set of analytical results from the anonymization techniques which do not take advantage of multiple paths to the failure presented in BBR [8].

Figure 3 exhibits the number of bits revealed, in percentage of the total input size. The percentage of the original input that remains unchanged is shown in the *residue* metric, presented in figure 4. Regarding the residue it should also be mentioned that the identification of the bits of the original input that remain unchanged, allows applications or error reporting tools to show users exactly what part of their original inputs will be included in the error report.

In the iDate test cases, the bug is at the end of a method

[4]http://idate.sourceforge.net/

that takes a profile of another user and compares it with the local profile. The only fault requirement is the use of different versions among the users. The comparison of the profiles is performed with a sequence of `ifs` whose outcome is irrelevant to the failure reproduction. The BBR's technique that only takes advantage of the original execution path, leaks a lot of information that is not relevant to the failure reproduction and, in this case, very sensitive data as, for example, the user's sexual orientation. As MultiPathPrivacy takes advantage of various possible paths to the point of failure, the improvement of privacy is significant.

The failure exercised in the Cruise Control test cases is activated when it receives the order to resume the cruise control speed without any previous activation. Thus, any sequence of input that has the order to resume the cruise control speed without previously having activated the command causes the application failure. The test cases were performed with an input that diminishes until the Cruise Control4 test case which only contain the resume command. Thus, the percentage of *bits of information revealed* will rise to reach 100% in the case where the input contains only essential information for the application failure. The *residue* has a similar growth. BBR's technique reveals 100% of the information in all the Cruise Control test cases because it reproduces exactly the original execution path that, in this case, only have one possible input.

In the Elevator test cases, failure occurs when a stop is requested for a floor number that is higher than the number of floors in the system. Similar to what was done in the Cruise Control test cases, the number of valid values introduced before introducing the invalid one, decrease along the test cases. This happens until Elevator3 test case that consists only in the invalid value. In this case, the results does not show a large improvement in privacy values between MultiPathPrivacy and BBR's technique, since the original path has a large domain of input which, by itself, guarantee a good level of privacy to the user.

C. Overhead

The use of deterministic replay system is constrained by the imposed overhead while monitoring the execution and processing the information to be included in the error report. In MultiPathPrivacy, particularly in the client component that is running on the user's machine, there are two crucial points to measure the MultiPathPrivacy's performance impact. The first is related to the monitoring process, during the normal application execution, in which MultiPathPrivacy only records the final line of code. Since this feature is achieved by surrounding the original application code with a `try/catch` block, the performance penalty is almost imperceptible. In all tests this cost is constant and is 12 ms per run.

The second point of analysis is the cost of systematic reproduction of execution paths to divide the set of possible execution paths is the ones that crash the application and those that don't. The component that makes the replay on the client, LEAP *replayer*, test the various possibilities of paths to the point of failure in order to collect the set of paths that reproduce the failure. This component has a constant initial cost of about 70 ms, which is added the time of replaying each of the possible paths. Table II shows the values for the overhead of running the three applications used in the evaluation.

Application	Original Execution	# Paths	Total Replay
Cruise Control	0.011s	343	4.15s
Elevator	0.023s	32	0.78s
iDate	0.191s	1260	294.30s

Table II
PATH REPLAYING OVERHEAD

To achieve the privacy levels presented in the last section is necessary to make the re-runs of all possible paths to the point of failure, shown in table II. The user's privacy is incremented every time that a replay of the application crashes and the corresponding path condition is less restrictive than the union of all previous successful replays. Although the high cost of reproduction time for all paths that pass through the point of failure, the user can define an acceptable level of privacy, from which there is no longer necessary to do more re-executions. Moreover, this computation can be performed during periods when the computer is idle.

D. Scalability

The search of several possible execution paths in an application, is performed, in the server component, with the technique of symbolic execution. This technique has a known scalability issues due to the combinatorial explosion of paths. To try to control this combinatorial explosion of paths, the symbolic execution performed in MultiPathPrivacy, is applied to previously chosen methods and not for the entire application. Still, in iDate, the heaviest test case, the symbolically executed method contains a set of `if` clauses which causes a rapid growth in the number of path possibilities. Table III shows the evolution of several execution results along the growth of the number of `if` clauses, namely the number of states visited, the number of instructions, the maximum memory used and time spent.

# if	States	Instructions	Max memory	Execution time
1	4	3027	77MB	2,01s
2	8	3101	77MB	2,09s
4	24	3285	77MB	2,20s
6	72	3549	303MB	3,03s
8	396	6033	381MB	9,07s
10	2016	16293	383MB	5min08s
11	4536	60393	383MB	26min89s
12	9576	103233	383MB	2h36min64s

Table III
EXECUTION RESULTS EVOLUTION WITH THE GROWTH OF THE NUMBER OF IF CLAUSES.

VI. Conclusions

This paper introduced MultiPathPrivacy, a system that provides fault replication with privacy for the user. MultiPathPrivacy stands out among the anonymization techniques of reference [8], [9] with the use of multiple execution paths in the failure's reproduction. Privacy guaranteed by these techniques depends solely on the domain size imposed by the conditions of the original path, that is, the number of inputs that satisfy the path conditions. With MultiPathPrivacy, the privacy level is enhanced not only by the domain size of the original path conditions, but also with the domain size of other failure leading path conditions.

To quantify the levels of privacy and analyze the full application of MultiPathPrivacy, we compare it with the solutions that take advantage on only one execution path. An experimental study was conducted with a total of eleven test cases in three different applications, one being used in the world real. The level of privacy was quantified using two metrics, namely *bits of information revealed* and *residue*. The first metric is a pure measure of entropy and the latter represents the percentage of the original input that remains unchanged after the anonymization. The results clearly show the benefit of using MultiPathPrivacy as it improved up to 87% in the both privacy metrics and, in the worst case, matched the state of the art results.

Although the evaluation presented shows limitations in terms of scalability, this approach can be extended in order to cope with scale limitations. One obvious approach is to perform the search of the path conditions of all possible execution flows not at the level of a whole application but using some modularity criterium, e.g. perform the analysis for each one of the listeners of an event based application. Another possibility, which we are currently exploring, is replacing the symbolic analysis done before deployment, during the development process, by a dynamic execution performed at the client device after the application crashes. In this alternative, the application is executed symbolically starting from the execution path that led to the failure and exploring alternative paths that may leak less user input data than the original path.

Acknowledgments

This work was partially supported by FCT (INESC-ID multi-annual funding) through the PIDDAC program funds, and by the European Union "FastFix" project (FP7-ICT-2009-5). Parts of this work have been performed in collaboration with other members of the Distributed Systems Group at INESC-ID, namely, Luís Rodrigues, Nuno Machado and João Matos.

References

[1] G. Candea, "Exterminating bugs via collective information recycling," *Dependable Systems and Networks Workshops*, vol. 0, pp. 200–204, 2011.

[2] M. Zhivich and R. Cunningham, "The Real Cost of Software Errors," *IEEE Security & Privacy*, vol. 7, no. 2, pp. 87–90, 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4812166 http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4812166

[3] B. R. Liblit, "Cooperative bug isolation," Ph.D. dissertation, University of California at Berkeley, 2004.

[4] K. Saeed, "It is that Dreaded Error Report: An Empirical Assessment of Error Reporting Behavior," in *Proceedings of the 2005 SIGHCI*, 2005. [Online]. Available: http://aisel.aisnet.org/sighci2005/7

[5] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. IEEE Computer Society, 2007, pp. 344–353. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.45

[6] J. Huang, P. Liu, and C. Zhang, "LEAP: lightweight deterministic multi-processor replay of concurrent java programs," in *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 207–216. [Online]. Available: http://portal.acm.org/citation.cfm?id=1882323

[7] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. Lee, and S. Lu, "Pres: probabilistic replay with execution sketching on multiprocessors," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 177–192. [Online]. Available: http://portal.acm.org/citation.cfm?id=1629593

[8] M. Castro, M. Costa, and J.-P. Martin, "Better bug reporting with better privacy," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 1, p. 319, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1353534.1346322

[9] J. Clause and A. Orso, "Camouflage: automated anonymization of field data," in *Proceeding of the 33rd international conference on Software engineering*, ser. ICSE '11. ACM, 2011, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985797

[10] P. Broadwell, M. Harren, and N. Sastry, "Scrash: a system for generating secure crash information," in *Proceedings of the 12th conference on USENIX Security Symposium*, vol. 12. USENIX Association, 2003, pp. 19–19.

[11] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, 1976.

[12] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *Proceedings of the 2008 international symposium on Software testing and analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 15–26.

[13] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, pp. 203–232, 2003.

[14] P. Louro, "Privacy in fault replication (original title: Privacidade em replicação de falhas)," Master's thesis, Instituto Superior Técnico, Technical University of Lisboa, 2011.

[15] R. Wang, X. Wang, and Z. Li, "Panalyst: privacy-aware remote error analysis on commodity software," in *Proceedings of the 17th conference on Security symposium*. USENIX Association, 2008, pp. 291–306. [Online]. Available: http://portal.acm.org/citation.cfm?id=1496711.1496731

[16] C. E. Shannon, "The mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 4, pp. 379–423, 623–656, 1948. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/9230594

[17] S. Mouchawrab, L. Briand, Y. Labiche, and M. Di Penta, "Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 161–187, 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5416729 http://www.computer.org/portal/web/csdl/doi/10.1109/TSE.2010.32

[18] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005. [Online]. Available: http://www.springerlink.com/index/PJ43552632155738.pdf

[19] Y. Ma, J. Offutt, and Y. Kwon, "MuJava: An automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/stvr.308/abstract