# REAP: Reporting Errors Using Alternative Paths[⋆]

João Matos, João Garcia, and Paolo Romano

INESC-ID / Instituto Superior Técnico

**Abstract.** Software testing is often unable to detect all program flaws. These bugs are most commonly reported to programmers in error reports containing core dumps and/or execution traces that frequently reveal users' private information without providing all necessary information for effective debugging. Hence, these mechanisms are sparsely used due to users' data privacy concerns. This paper presents REAP, a new fault replication method, which allows for enhancing privacy protection while still providing software developers with the 'steps-to-reproduce" errors. REAP uses symbolic execution and randomized search heuristics to identify alternative execution paths leading to an observed error. We evaluated REAP using a testbed including real bugs of popular, large scale applications. The results show the high effectiveness of REAP in anonymizing user input: on average, REAP reveals only 16.78% of the bits in the original input, achieving an average residue (the number of common characters in the original and anonymized input) of 15.07%. Our evaluation also highlights that REAP significantly outperforms state of the art techniques in terms of achieved privacy and/or scalability.

**Keywords:** Software Bugs, Error Reporting, Fault-Replication, Privacy.

## 1 Introduction

It is common for software errors to manifest themselves after the software is released and persist long after that [1], despite more than half of the resources in a typical development cycle being invested in testing and bug fixing. Software bugs represent several billion dollars per year worth of maintenance costs in Europe and in the US alone [2]. Currently, the most popular tools to provide developers with information about application crashes (e.g. [3–5]) are error-reporting tools. These tools aim to allow software vendors to fix bugs in a timely manner. However, error reports usually include solely partial snapshots of the memory, stack traces of the failed process and a textual description of the faulty scenario, which is often insufficient to reproduce the error [6, 7]. Fault replication mechanisms address the shortcomings of classical error reports, by allowing engineers to reproduce, at the development site, a faulty execution taken place at the client side. These mechanisms monitor target applications on client devices in order to gather enough information for execution reproduction, while imposing the least overhead possible. Numerous fault-replication mechanisms have been

[⋆] This work was supported by national funds through FCT - Fundação para a Ciência e Tecnologia - under project PEst-OE/EEI/LA0021/2013, and by GreenTM project (EXPL/EEI-ESS/0361/2013).

developed and are becoming more capable of efficient application monitoring and successful bug reproduction ([8–10] to name a few). Unfortunately, privacy and security concerns have prevented widespread adoption of many of these techniques and, because they rely on user participation, have ultimately limited their usefulness [11]. In fact, whether the user is working on a confidential document or has typed in personal information, sensitive private information is likely to be included either in the memory snapshot taken to generate an error report or in the non-deterministic sources logged by fault replication mechanisms [12].

A promising approach aimed at tackling these privacy concerns is based on the idea of obfuscating sensitive information inserted while ensuring the reproduction of the faulty execution ([11–13]). These mechanisms use symbolic execution (e.g. [14]) in order to derive a set of logical constraints of the user input, called *path condition* [15], that ensures the application will re-execute along the same execution path that previously led to failure. Alternative inputs, which reproduce the bug can then be drawn from the set of all inputs satisfying the identified path condition. This approach was shown to have the potential to achieve high obfuscation levels since large portion of the input data can often be replaced by alternative values derived from less constrained symbolic values. However, the degree of obfuscation achievable by these techniques is directly dependent on the restrictiveness of the path condition's constraints (i.e. on the cardinality of the set of inputs that match a given constraint), which can be critically affected by the application's structure and bug placement in the code.

In this paper we propose REAP (Reporting Errors using Alternative Paths), a novel approach based on the idea of increasing the degree of obfuscation by exploiting the presence of alternative execution paths leading to the same failure. REAP relies on symbolic execution techniques, and on lightweight search heuristics that perform bounded-depth detours from the original execution path in order to identify alternative, failure inducing paths (and their corresponding alternative user inputs). We provide a theoretical analysis of the search heuristics employed by REAP, establishing a conservative upper bound on the information leakage that it can achieve and the information that an attacker can derive on the original user input. We present the results of an extensive experimental analysis based on 6 publicly available applications, which includes popular, large scale software projects and privacy-sensitive applications from the financial and online dating domains. REAP's evaluation assesses the feasibility of the proposed solution in realistic settings, and quantifies the obfuscation quality enhancements achievable in comparison with state of the art solutions. The results show that, contrasted with state of the art solutions analyzing solely the conditions of the original execution path, REAP can achieve, with comparable execution times, up to an 83.22% average reduction in revealed input data. Furthermore, REAP can identify alternative inputs in a matter of minutes with large scale applications.

This paper is organized as follows. Section 2 overviews existing obfuscation mechanisms and discusses their main strengths and limitations. Section 3 presents the REAP system. We evaluate the proposed system in Sec. 4 before presenting some concluding remarks.

## 2 State of the Art and Motivations

### 2.1 Final Application State Error Reporting

Initial approaches to automatic error report, such as Windows Error Reporting [4] and Mozilla Crash Report [5] involved mainly information collected at the end of a failed program execution. When an application crashes, the error reporting system gathers information uncritically from the state of the process at the moment of the crash and submits it as an error report, if authorized by the user. Two major disadvantages of these methods stand out: *i*) there is no filtering of the submitted information regarding users' privacy preservation, which means that sensitive information may end up being incorporated in the dump of the application state performed upon the occurrence of the bug [12]; *ii*) the generated report does not provide any historical information on how the error was reached, which typically makes the reproduction of the bug a complex and time consuming task [6, 7].

One of the first systems to attempt to filter user private information from error reports was Scrash [16]. Applications have all their sensitive data marked as such during development, and allocated in a specially reserved area of memory. When an error report is submitted for a Scrash enabled application, all the sensitive variables are removed. This approach has three main problems. First, it requires access to an application's source code. Second, it assumes that the application programmers are trustworthy and will mark all sensitive data as such. And finally, error reports that have been amputated of relevant data may not allow for the full replay of the original error.

### 2.2 Input Anonymization in Fault Replication Systems

Fault replication systems that transmit the user input to the maintenance site arguably raise even larger privacy concerns. Two main approaches have been proposed to identify anonymized, failure-inducing inputs: input minimization [17] and path condition analysis [11, 12, 18].

Input minimization techniques [17] were originally designed to speed-up testing/debugging and attempt repeated random removals of input chunks, in order to identify input fragments that are irrelevant for the reproduction of the bug. By purging irrelevant inputs, these techniques can enhance privacy. However, as discussed in previous works [11, 12], due to their purely random nature, input minimization techniques typically fail in frequent scenarios in which valid inputs must respect precise structural conditions. (e.g. a credit card number must be composed of exactly 16 digits satisfying the Luhn checksumming algorithm; XML documents must comply with a defined structure).

Approaches based on path condition analysis [11, 12, 18] overcome these limitations, by reasoning on the logical constraints imposed by the conditional branches that were taken during a failure-inducing execution, i.e. its path condition. In other words, the logical restrictions imposed by a path condition delimit the domain from which input values can be chosen and still trigger the same error.

```
1: int age, n=0; /*bug source*/
2: boolean isMale, isMarried;
3: double score=1;
4: read from input: age, isMale, isMarried;
5: if (age>25)
6:      score = score * 1.5;
7: else
8:      score = score / 2;
9: if (isMale)
10:     score = score * 2;
11: else
12:     score=score / 2;
13: if (isMarried)
14:     score = score * 2;
15: else
16:     score = score / 2;
17: score = score / n /*divide by 0*/
```

**Fig. 1.** Example code excerpt

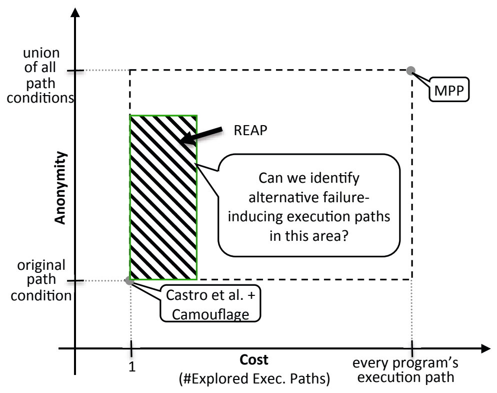
union of all path conditions
MPP
REAP
Anonymity
Can we identify alternative failure-inducing execution paths in this area?
original path condition
Castro et al. + Camouflage
1
Cost (#Explored Exec. Paths)
every program's execution path

**Fig. 2.** Trade-off explored by REAP

Therefore, the degree of obfuscation attained by these approaches [11, 12, 18] is critically affected by the restrictiveness of the logical clauses in a path condition. The two main metrics to evaluate privacy in this context [11, 12] are the *number of leaked information bits* (henceforth called *leakage*) and the *residue*. The leakage of a particular path condition is calculated as $-\log_2(\alpha)$, where $\alpha$ is the fraction of the domain of the application input variables that satisfy the path condition. The residue is a more intuitive metric defined as the number of input characters that remain unchanged after the anonymization process.

The code excerpt in Figure 1 is used to motivate and illustrate the behavior of REAP. We note that the code excerpt exhibits a trivial bug (division by 0 caused by a wrong initialization of variable $n$ in line 3, which manifests itself in line 17). The bug could be easily detected using classic debugging tools. However, despite its simplicity, the example clearly highlights the potentialities of REAP and the limitations of the two existing approaches. Now let us assume that the user input: $age = 26$, $isMale = true$, and $isMarried = false$. The path condition derived from the execution with this data yields the constraints:

$$age \in [25, MaxInt] \ \wedge \ isMale \ \wedge \ \overline{isMarried} \tag{1}$$

In such a scenario, the age input by the user can be (partially) obfuscated by replacing it with any value larger than 25. The remaining input values, on the other hand, will have to be fully disclosed. It should be noted that in the program of Figure 1, it is actually possible to achieve total input anonymization, given that the bug manifests in all possible execution paths, and, hence, independently from the value of the user input. MPP [19] is, to the best of our knowledge, the only system that attempts to exploit the presence of multiple failure-inducing execution paths in order to maximize the obfuscation level of a bug report.

By considering the disjunction of the path conditions of *all* execution paths leading to a bug, MPP can achieve, at least for small-scale programs, the theoretical lower bound on information leakage, identifying *all* the possible inputs that replay the bug. Unfortunately, MPP suffers from severe scalability limitations for two main reasons. MPP relies on an off-line reachability analysis that performs a symbolic execution of the program, and produces as output, for all lines of code, a path condition and a triggering input for *all* the execution paths that traverse that line of code. As demonstrated by the experimental data presented in MPP's paper [19], and as confirmed by our experimental evaluation, the costs associated with MPP's off-line reachability analysis are prohibitive for applications other than small scale ones. Also, as not all the execution paths identified during the symbolic execution may actually trigger the bug, the client needs to re-execute all of them, in order to verify which subset of the paths actually reproduces the error. This can be quite inefficient especially if the bug is located in a line of code that happens to be reachable through a high number of paths.

In the considered code example, MPP would generate 8 path conditions, each one associated with different combinations of the three tests on the input variables. As all of these paths lead to the bug, the disjunction of their path conditions yields a total relaxation of the constraints on the input variables, and achieves perfect anonymization. Unfortunately, the price for attaining such a boost on input obfuscation grows exponentially with the number of tests on different input variables contained by the program.

As depicted in Fig. 2, REAP seeks an innovative balance between efficiency and anonymity in the design space of privacy preserving fault replication schemes. At the extreme of lowest anonymity are systems like in [11, 12, 18], which explore only the original execution path. On the other hand, by exploring all execution paths leading to the observed point of crash, MPP may provide the maximum possible anonymity, but suffer of severe scalability limitations. REAP strikes a balance between these two extremes, by taking advantage from alternative failure-inducing execution paths, while using scalable search heuristics that ensure its practicality even in large-scale, complex programs.

## 3 REAP

This section presents an overview of REAP's framework, used to generate search heuristics aimed at identifying alternative failure-inducing execution paths. It is also discussed the anonymization capabilities of REAP.

### 3.1 Overview of the System

The various stages of execution of REAP are illustrated by the diagram in Figure 3, and described in the following.

**Original Input Anonymization Phase.** Similarly to existing fault-replication systems [11, 12, 19], REAP relies on automatic code instrumentation to log user inputs in a transparent fashion. When an application failure $f$ is detected, REAP re-executes symbolically the application feeding it with the original failure-inducing input $I$ (just as in the systems in [11, 12]). The Original Input Anonymization

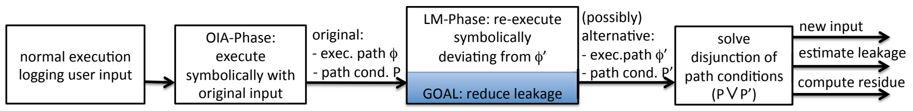
normal execution logging user input
OIA-Phase: execute symbolically with original input
original:
- exec. path ϕ
- path cond. P
LM-Phase: re-execute symbolically deviating from ϕ'
GOAL: reduce leakage
(possibly) alternative:
- exec.path ϕ'
- path cond. P'
solve disjunction of path conditions (P∨P')
new input
estimate leakage
compute residue

**Fig. 3.** Architectural Overview of REAP

(OIA) phase pursues a twofold goal: *i*) identifying the sequence of program statements composing the original failure-inducing execution path, denoted as $\phi$; *ii*) computing the path conditions, $P$, associated with $\phi$.

**Leakage Minimization Phase.** Next, REAP executes what we called *leakage minimization* (LM) phase. In this phase REAP relies on randomized depth-bounded search heuristics that aim to identify an alternative failure-inducing execution path $\phi'$ by performing controlled detours from $\phi$. To this end, we introduce a flexible search heuristics framework, which allows not only to control the duration/extensiveness of the search phase, but also to customize the behavior of the search algorithm, i.e., the logic controlling the selection of the detouring points and the trajectories to explore once a detour is ongoing. In this work we show how REAP's search framework can be used to derive two alternative heuristics for which we prove a fundamental property: if REAP identifies an alternative path $\phi'$, it guarantees that no attacker can deterministically deduce $\phi$, even if she is aware of the topology of the full execution graph and has unbounded processing capabilities. Beyond that, if such an attacker performed a probabilistic analysis of every possible failure-inducing path, it could only deduce that the most likely original execution path coincides with that output by REAP (i.e., $\phi'$), hence effectively concealing $\phi$. Section 3.3 provides a theoretical analysis of REAP's anonymization capabilities.

**Privacy Evaluation and Report Submission.** Once $\phi'$ is obtained, REAP determines a feasible value for the input $I'$ that triggers the execution path $\phi'$ by finding a solution to the corresponding path condition $P'$. Further, REAP computes the residue associated with $I'$, and derives a conservative lower bound for the attained leakage level. Finally, the user is presented with the anonymized input, along with the corresponding leakage and residue values, and is asked to authorize the transmission of the bug report to the maintenance site.

## 3.2 Search Heuristics Framework

As we have already mentioned, REAP searches for alternative failure-inducing execution paths by performing detours of bounded length from the original faulty execution path $\phi$. Before detailing the algorithms employed by REAP to this end, we introduce how an execution path is modeled in REAP.

REAP associates with the execution path $\phi$ of a program a directed acyclic graph where each node of the graph represents a sequence of statements comprised between two subsequent conditional tests on some input-dependent variable. The graph is built dynamically, during the symbolic execution of $\phi$, adding a node to the graph (and connecting it to the previously generated node) every time that a branch of an input-dependent test is taken. Whenever a node is added to the graph, this is also labeled using the following triple: a location identifier composed by line of code and class signature; the current stack trace; the value of the current iteration of any cycle within which the node is being executed. This simple scheme allows us to avoid aliasing problems, ensuring that if a program statement is executed in two different execution contexts, two unique identifiers will be attributed to it. As in typical symbolic execution engines [20], an execution path is modeled assuming that each logical test generates only two edges[1], hence the execution graph is a binary tree.

We can now present the framework used to generate the search heuristics employed by REAP. The framework is embodied by the function $\phi$SEEKER, whose pseudocode is shown in Algorithm 1. This function encapsulates the logic of a generic search heuristic that, given the original execution path $\phi$ and a fault $f$[2], returns the path condition of a possible alternative failure-inducing execution path $\phi'$. The behavior of $\phi$SEEKER is customizable via the following parameters:

- *numDetours:* the total number of detours the search heuristic should attempt.
- *maxDetourLength:* the maximum depth that the search heuristic can traverse after having performed a detour and before joining back the original path;
- *maxAttempts:* the maximum number of times that the LM-phase can be run;

and via the following two functions, whose implementation allows to flexibly derive a wide range of alternative search algorithms:

- DETOURSSELECTOR takes as input the original path $\phi$, and the total number of detours that should be attempted, *numDetours*, and returns a set of *numDetours* nodes in $\phi$ from which $\phi$SEEKER should attempt a detour;
- PICKCHILD is used whenever a detour is being performed, to determine which of the two branches outgoing from a node (passed as input parameter) should be explored next. Both functions accept also as input parameter the identifier of the current search attempt, in order to allow the definition of adaptive policies whose behavior evolves across different attempts.

The heuristics' behavior is fully specified by defining how they implement the functions DETOURSSELECTOR and PICKCHILD, as well as they set the *numDetours* parameter. The parameters $maxDetourLength$ and $maxAttempts$ are used as tuning knobs to control, respectively, the radius of the search, and the maximum duration of the search. At each step of the search, the current node is executed symbolically and the corresponding logical constraint is added to the path condition $P$ that identifies the domain of feasible input values that are able to replay the current execution path. The logical constraint for the first

[1] This simplifies reasoning on the execution graph, while still allowing capturing arbitrarily complex branching structures.

[2] We assume that faults are observable and uniquely identifiable as in [11, 12, 19].

**Algorithm 1.** Pseudocode defining the family of algorithms used to identify alternative paths

```
1  function $\phi$SEEKER
   Input parameters:
       ExecPath $\phi$;
       Fault $f$;
       int $numDetours$, $maxDetourLength$, $maxAttempts$;
       function Node PICKCHILD(Node $n$, int $attempt$);
       function Set<Node> DETOURSSELECTOR(ExecPath
   $\phi$, int $numDetours$, int $attempt$);
   Output parameter:
       PathCondition;

2  begin
3      for int $currAtt$=0; $currAtt < maxAtt$; $currAtt++$ do
4          Set<Node> $detours = \emptyset$;
5          PathCondition $P = \emptyset$;
6          $detours$ = DETOURSSELECTOR($\phi$, $numDetours$, $currAtt$);
7          if ( FORWARD($\phi$.getFirstNode(),$P$) $\wedge$ $P \neq \phi$.getPathCondition() ) then
               // an alternative failure-inducing path was found
8              if $currAtt$ == 1 then
9                  return $P$;
10             else
11                 return with probability 0.5 either $P$ or $\phi$.getPathCondition();
12     return $\phi$.getPathCondition();

13 boolean FORWARD(Node $n$, PathCondition $P$)
14 begin
15     if $n$ == null then
           return false;
16     execute $n$ symbolically;
17     add to $P$ the logical constraint of $n$;
18     if $n$ reproduces $f$ then
           return true;
19     Node $next$, $checkp$, $current$;
20     $checkp$ = $next$ = the successor of $n$ that lays on the original path $\phi$;
21     if $n \in detours$ then
22         $current$ = the successor of $n$ that does not lay on the original path $\phi$;
23         $next$ = DETOUR($current$, $maxDetourLength$, $P$);
24         if $next$ == null then
               $next$ = $checkp$; // detour failed, continue along the original path $\phi$
25     return FORWARD($next$, $P$);

26 Node DETOUR(Node $n$, int $bound$, PathCondition $P$)
27 begin
28     if $bound$ == 0 then
           return null;
29     execute $n$ symbolically;
30     Node $next$ = PICKCHILD($n$, $currAtt$);
31     if $next \in \phi$ then
           // the detour has re-joined the original path $\phi$
32         add to $P$ the logical constraints of this detour;
           return $next$;
33     return DETOUR($next$, $bound$-1, $P$);
```

starting node of the program is void, but, for every other node $n$, it is equal to the logical condition imposed by the edge connecting $n$'s predecessor to $n$.

Next, if the current node has been selected for a detour (line 21), a detour attempt is performed using the DETOUR function. This function implements a bounded-depth search in which, at each step, the next node to be explored is selected by means of the PICKCHILD method. If the detour joins back the original path (line 31), the path condition of the detour is added to that of the current execution path. Otherwise, if the detour reaches the upper bound on its length (*maxDetourLength*) without joining the original path, the detour attempt is aborted and the exploration proceeds along the original path (line 24).

The FORWARD function can terminate either because it reaches the same crash point as $\phi$ and does not reproduce $f$ (line 15) - which can happen if one or more nodes of $\phi$, required to reproduce $f$, were detoured - or because it replays $f$ (line 18). Note that in the latter case, FORWARD may fail all the detour it attempts and return the original path. This case is detected in line 7, where it is accounted as a failed attempt. In case of successful identification of an alternative failure-inducing path, $\phi$SEEKER behaves differently depending on whether this is the first attempt or not. In the former case, the corresponding path condition $P$ is returned. If REAP performs multiple attempts to find path $\phi'$ it may create a bias towards $\phi$. For example, if only two failure inducing paths exist and $maxAttempts = \infty$, REAP eventually finds the alternative path with probability 1. Consequently the original path could be deduced deterministically by an attacker who knows REAP's behavior. To cope with this issue, if REAP requires more than one attempt to find an alternative path, it returns $\phi$ with probability 0.5 (line 11). As we will discuss in Section 3.3 this allows effectively concealing the original path $\phi$ in case an alternative path $\phi' \neq \phi$ is returned by $\phi$SEEKER. Finally, if no failure-inducing path is identified after *maxAttempts* attempts, $\phi$SEEKER simply returns $\phi$.

Below we describe two different search algorithms, which we called Bounded Random Walk (REAP-BRW) and Bounded Adaptive Greedy (REAP-BAG).

*REAP-BRW: Bounded Random Walk.* This algorithm has a similar behavior to a random walk, within the radius bounded by *maxDetourLength* around $\phi$. The value of *numDetours* is picked at random between 0 and the length of $\phi$. Further, DETOURSSELECTOR selects *numDetours* nodes in $\phi$ as the source of a detour with uniform probability. Finally, the function PICKCHILD returns a child node at random, also with equal probability.

*REAP-BAG: Bounded Adaptive Greedy.* A logical test made on a set of input variables generates two edges that divide the input domain (of these variables), usually in a not equal way. This algorithm is biased to pick the edge outgoing from a node, whose path condition is satisfied by the largest number of input values (i.e., associated with the least restrictive path condition), a *broad edge*. We refer to the edge associated with the smaller part of the domain as *narrow edge*. This heuristic tends to choose *broad edges* over *narrow edges*, although with an adaptive probability, which decreases as the number of attempts performed so far increases. For this algorithm, the function SORTCHILD returns the child node that encompasses the largest fraction of the input domain. The SORTCHILD function implements the adaptive greedy policy, by selecting a broad edge from the currently visited node (automatically selected for the detour as $numDetours = |\phi|$) with probability $P(B)$:

$$P(B) = \frac{t+1}{2t} \tag{2}$$

where $t$ is the attempt being performed, and a narrow edge with the complementary probability $P(N) = 1 - P(B)$. This ensures that in the first iterations REAP-BAG will attempt with higher probability to follow the least restrictive execution paths, while converging the behavior towards the REAP-BRW heuristic as the number of attempts grows. We note that this heuristic is inspired to analogous policies used in the context of reinforcement learning problems to explore the trade-off between exploration and exploitation in face of uncertainty in [21]. In the cases where the input domain is divided equally, the edges are chosen with 0.5 probability, like in REAP-BRW. The DetoursSelector function selects the nodes in $\phi$ to be the source of a detour, with the probability given by equation 2.

### 3.3 Privacy

In this Section we analyze the privacy properties of *REAP-BRW* and *REAP-BAG*.

**Preliminary notations.** We denote with $\mathcal{F}$ the set of all failure inducing paths and with $\mathcal{F}(\phi', MDL)$ the set of all failure-inducing paths from which the execution path $\phi'$ could be obtained via detours of maximum length equal to $MDL$. Further, we denote with $i$ the original input that triggered the bug, and with $\mathcal{I}(\phi)$ the set of inputs triggering an execution path $\phi$. Finally, we denote respectively $P(BRW \rightarrow \phi')$, $P(BAG \rightarrow \phi')$, the probability that REAP-BRW, REAP-BAG output an input associated with the failure-inducing path $\phi'$ starting from the failure-inducing path $\phi$. When we refer to both REAP's variants we write, instead, $P(R \rightarrow \phi')$.

**Proof overview.** We demonstrate that the original path, denoted as $\phi$, cannot be deduced from the path output by REAP, denoted as $\phi'$. To do so, we first demonstrate that if REAP outputs a path $\phi'$, then $\phi'$ is the execution path in $\mathcal{F}(\phi', MDL)$ that is the most likely of being the original path. Next we discuss why, in case REAP outputs an alternative path $\phi' \neq \phi$, the information leakage of $\phi \cup \phi'$ and can be used as an upper bound of the information leakage reached by REAP. This result allows us to derive a methodology to quantify and report to end-users the information leakage allowed by REAP. Before presenting the proofs, we introduce some preliminary remarks.

*Remark 1.* In order for REAP to be application independent, its privacy guarantees (including the measurements of both leakage and residue) rely on the assumption of pure entropy, just like in all previous work [11, 12, 19]. Hence, we assume no *a priori* knowledge on the input structure nor on any information that can be deduced or contextualized in the program semantics.

*Remark 2.* Let $\phi^*$ be the original failure-inducing path in $\mathcal{F}(\phi', MDL)$ (note that this set also includes $\phi^* = \phi'$). We denote with $C(\phi^*, \phi')$ the set of edges in common between $\phi^*$ and $\phi'$, and with $D(\phi^*, \phi')$ the set of edges present in $\phi^*$ and not in $\phi'$. The latter set contains the edges obtained when REAP performs a detour from $\phi^*$, whereas the edges in $C(\phi^*, \phi')$ are obtained whenever a node of $\phi^*$ is not selected to perform a detour, or when a detour attempt starting from that node fails. Finally $|C(\phi^*, \phi')| + |D(\phi^*, \phi')| = |\phi'|$.

*Remark 3.* Both REAP-BRW and REAP-BAG, when executed with $MDL = d$ starting from an execution path $\phi$, can only identify alternative paths $\phi'$ such that each sub-path (i.e., sequence of consecutive edges) $s_i \in D(\phi, \phi')$ has length at most $d$. This allows us to provide a more rigorous definition of the set of alternative failure-inducing paths identifiable starting from a path $\phi$, which we denoted as $\mathcal{F}(\phi, d)$: $\phi^* \in \mathcal{F}(\phi, d) \Rightarrow \forall s_i \in D(\phi, \phi^*)\ |s_i| \leq d$.

*Remark 4.* Since we are assuming that the only source of non-determinism is the user input, then, given two execution paths $\phi$ and $\phi'$ where $\phi \neq \phi'$, it follows that, given two inputs[3] $i \in \mathcal{I}(\phi)$ and $i' \in \mathcal{I}(\phi')$, they must differ by at least one bit. Hence, $\mathcal{I}(\phi) \cap \mathcal{I}(\phi') = \emptyset$.

**Theorem 1.** *Assume REAP-BRW is provided with the execution path $\phi$ as input and that it returns a (possibly different execution path) $\phi'$. Then among all paths $\phi^* \in \mathcal{F}(\phi', d)$, no path has higher probability of being the original path than $\phi'$. Formally:* $\phi' \in \underset{\phi^* \in \mathcal{F}(\phi', d)}{\operatorname{argmax}}\ P(BRW \to \phi' | i \in \mathcal{I}(\phi^*))$

*Proof.* For REAP-BRW to generate path $\phi'$ starting from path $\phi^*$, with $\phi^* \neq \phi'$, in one of the *maxAttempts* attempts it performs the following must happen:

1. for all edges $c \in C(\phi^*, \phi')$, REAP-BRW must either i) not detour from the original path $\phi^*$, or ii) detour from the original path and fail the detour attempt. As the start node, say $n_c$, of an edge $c \in C(\phi^*, \phi')$ is also in the original path $\phi^*$, when REAP-BRW encounters $n$, it decides whether to detour with probability 0.5. Conversely, the probability of failing a detour attempt from node $n$ depends on the actual topology of the execution graph of the program, but it is independent from the original path $\phi^*$; we denote this probability as $P_{fd}(n_c)$ and assume it unknown in the following. Overall, the probability for REAP-BRW to generate all the edges $n_c \in C(\phi^*, \phi')$ starting from $\phi^*$ is:
$$\prod_{n_c \in C(\phi^*, \phi')} 0.5 + P_{fd}(n_c)$$
2. when it encounters the starting node, say $n$, of every edge $d_i \in D(\phi^*, \phi')$, REAP-BRW must select (between the two edges outgoing from $n$) the edge $d_i \in \phi'$. As REAP-BRW picks an edge during a detour with probability 0.5, the probability for REAP-BRW to generate the edges in $D(\phi^*, \phi')$ is $0.5^{|D(\phi^*, \phi')|}$.

[3] Recall that when we refer to an input $i \in \mathcal{I}(\phi)$, we mean the entire string of bytes provided as input to trigger the execution path $\phi$.

Hence, the conditional probability that REAP-BRW identifies path $\phi'$ from any path $\phi^* \in \mathcal{F}(\phi', d)$ in a single attempt, given that the original user input was associated with $\phi^*$ is:

$$P(BRW \to \phi' | i \in \mathcal{I}(\phi^*)) = 0.5^{|D(\phi^*,\phi')|} \cdot \prod_{n_c \in C(\phi^*,\phi')} 0.5 + P_{fd}(n_c) \qquad (3)$$

It is straightforward to observe that:

$$\phi' \in \underset{\phi^* \in \mathcal{F}(\phi',d)}{\operatorname{argmax}} \ P(BRW \to \phi' | i \in \mathcal{I}(\phi^*))$$

as i) the cardinality of $|C(\phi^*, \phi')|$ is maximum when $\phi^* = \phi'$, and ii) $P_{fd} \geq 0$. Hence, no path in $\mathcal{F}(\phi', d)$ is more likely to be the original path than $\phi'$, if REAP-BRW outputs $\phi'$ in a single attempt.

On the other hand, if REAP identifies an alternative path $\phi' \neq \phi$ using more than one attempt, it outputs, with probability 0.5, either $\phi$ or $\phi'$. This guarantees that no path in $\mathcal{F}(\phi', d)$ has higher probability of being the original path than $\phi'$. □

**Theorem 2.** *Assume REAP-BAG is provided with the execution path $\phi$ as input and that it returns a (possibly different execution path) $\phi'$. Then among all paths $\phi^* \in \mathcal{F}(\phi', d)$, no path has higher probability of being the original path than $\phi'$. Formally:* $\phi' \in \underset{\phi^* \in \mathcal{F}(\phi',d)}{\operatorname{argmax}} \ P(BAG \to \phi' | i \in \mathcal{I}(\phi^*))$

*Proof.* The proof structure is analogous to the one of Theorem 1, so only a sketch of proof is provided for space constraints. Consider the set of edges in $C(\phi^*, \phi')$, and denote with $B(C(\phi^*, \phi'))$, resp. $N(C(\phi^*, \phi'))$, the set of broad, resp. narrow, edges in $C(\phi^*, \phi')$. Also, denote with $E(C(\phi^*, \phi'))$ the set of edges that are neither broad, nor narrow - which we call *even* edges. Using the same arguments employed in the previous theorem, one can compute the probability that REAP-BAG generates all the edges $n_c \in C(\phi^*, \phi')$ starting from $\phi^*$, denoted as $P_C(BAG \to \phi' | i \in \mathcal{I}(\phi^*))$, as:

$$\prod_{n_c \in E(C(\phi^*,\phi'))} 0.5 + P_{fd}(n_c) \prod_{n_c \in B(C(\phi^*,\phi'))} P(B) + P_{fd}(n_c) \prod_{n_c \in N(C(\phi^*,\phi'))} P(N) + P_{fd}(n_c)$$

and the probability $P_D(BAG \to \phi' | i \in \mathcal{I}(\phi^*))$ of yielding the edges in $D(\phi^*, \phi')$:

$$0.5^{|E(D(\phi^*,\phi'))|} \cdot P(B)^{|B(D(\phi^*,\phi'))|} \cdot P(N)^{|N(D(\phi^*,\phi'))|}$$

The probability $P(BAG \to \phi' | i \in \mathcal{I}(\phi^*))$, which is equal to:

$$P_C(BAG \to \phi' | i \in \mathcal{I}(\phi^*)) \cdot P_D(BAG \to \phi' | i \in \mathcal{I}(\phi^*)) \qquad (4)$$

is maximum for $\phi^* = \phi'$, since $\forall \phi^* \in \mathcal{F}(\phi', d)$ with $\phi^* \neq \phi'$ it must be that $|C(\phi', \phi')| > |C(\phi^*, \phi')|$.

When considering scenarios in which an alternative path $\phi'$ is output after multiple attempts by REAP-BAG, the same considerations valid for REAP-BRW also apply to REAP-BAG. □

**Theorem 3.** *If REAP finds an alternative path $\phi'$ starting from a different path $\phi$, the information leakage is at most equal to that computed by considering the logical disjunction of the path conditions associated with $\phi$ and $\phi'$.*

*Proof.* Assume that an attacker was provided with the correct knowledge that, among all the paths in $\mathcal{F}(\phi', d)$, the actual original path may only be either $\phi$ or $\phi'$. In this case, the uncertainty of the attacker is smaller than if she had to select among the entire set of paths in $\mathcal{F}(\phi', d)$ (as, in general, the paths in this set may have a non-null probability of being the original path). The uncertainty of this scenario is therefore a lower bound on the actual uncertainty of the attacker. Hence, the leakage results that we derive in the following represent a consistent upper bound on the actual leakage allowed by REAP.

Given that we are assuming that all inputs are equiprobable, and that we are only considering the paths $\phi$ and $\phi'$, it follows that the probability that the user original input lies on path $\phi$, $\phi'$, denoted, resp., as $P(i \in \mathcal{I}(\phi))$, $P(i \in \mathcal{I}(\phi'))$, is:

$$P(i \in \mathcal{I}(\phi)) = \frac{|\mathcal{I}(\phi)|}{|\mathcal{I}(\phi \cup \phi')|},\ P(i \in \mathcal{I}(\phi')) = \frac{|\mathcal{I}(\phi')|}{|\mathcal{I}(\phi \cup \phi')|} \tag{5}$$

where we denoted with $|\mathcal{I}(\phi)|$ the cardinality of the input domain associated with $\phi$. The unconditional probability for both variants of REAP to output a failure inducing path $\phi'$ starting from a path $\phi^* \in \mathcal{F}(\phi', d)$ can hence be computed as:

$$P(R \to \phi' \wedge i \in \mathcal{I}(\phi^*)) = P(R \to \phi' | i \in \mathcal{I}(\phi^*)) \cdot P(i \in \mathcal{I}(\phi^*)) \tag{6}$$

The attacker can compute the probability that $\phi$ is the original path given that REAP outputs $\phi'$, denoted as $P(i \in \mathcal{I}(\phi) | R \to \phi')$, as follows:

$$P\ (i \in \mathcal{I}(\phi) | R \to \phi') = \tag{7}$$

$$= \frac{P(R \to \phi' \wedge i \in \mathcal{I}(\phi))}{P(R \to \phi' \wedge i \in \mathcal{I}(\phi)) + P(R \to \phi' \wedge i \in \mathcal{I}(\phi'))} = \tag{8}$$

$$= \frac{P(R \to \phi' | i \in \mathcal{I}(\phi)) \cdot P(i \in \mathcal{I}(\phi))}{P(R \to \phi' | i \in \mathcal{I}(\phi)) \cdot P(i \in \mathcal{I}(\phi)) + P(R \to \phi' | i \in \mathcal{I}(\phi')) \cdot P(i \in \mathcal{I}(\phi'))}$$

where, in order to derive Eq. 8 from Eq. 7, we have exploited Remark 4.

Since by Eq. 4 (for REAP-BAG) and Eq. 3 (for REAP-BRW) we have that $P(R \to \phi' | i \in \mathcal{I}(\phi)) \leq P(R \to \phi' | i \in \mathcal{I}(\phi'))$, we can obtain an upper bound for Eq. 7 by replacing in its denominator $P(R \to \phi' | i \in \mathcal{I}(\phi'))$ with $P(R \to \phi' | i \in \mathcal{I}(\phi))$, and simplifying the expression using Eq. 5:

$$P(i \in \mathcal{I}(\phi) | R \to \phi') \leq \frac{P(i \in \mathcal{I}(\phi))}{P(i \in \mathcal{I}(\phi)) + P(i \in \mathcal{I}(\phi'))} = \frac{|\mathcal{I}(\phi)|}{|\mathcal{I}(\phi)| + |\mathcal{I}(\phi')|} \tag{9}$$

Finally, for the attacker to correctly guess the actual user input, in addition to identifying that the original path was not $\phi'$ but $\phi$ (whose probability is given by Eq. 7), she needs to pick the correct input among those in $\mathcal{I}(\phi)$. Since we are assuming that inputs are equiprobable, the latter probability, which we note $P$(right input in $\mathcal{I}(\phi)$ is guessed), is simply $|\mathcal{I}(\phi))|^{-1}$ hence:

$$\begin{aligned}
&P\,(\text{original input is guessed}|R \to \phi') = \\
&= P(i \in \mathcal{I}(\phi)|R \to \phi') \cdot P(\text{right input in } \mathcal{I}(\phi) \text{ is guessed}) \leq \\
&\leq \frac{1}{|\mathcal{I}(\phi))| + |\mathcal{I}(\phi'))|} \qquad (10)
\end{aligned}$$

Recalling that, by Remark 4, $\mathcal{I}(\phi) \cap \mathcal{I}(\phi') = \emptyset$, the claim follows. □

## 3.4 Prototype Implementation

We implemented REAP for applications written in the Java language[4]. This tool has three main components: the execution monitor, the symbolic execution engine and the anonymizer. The execution monitor instruments the compiled Java application using the SOOT [22] bytecode instrumentation tool in order to log all user input in a transparent fashion. Note that, in order to ensure deterministic error replay, one should log all sources of non-determinism of the program, and not solely user input. On the other hand, dealing with other sources of non-determinism is out of the scope of the REAP system for the following two main reasons: *i*) different types of non-deterministic sources could be tackled using dedicated solutions aimed at supporting deterministic replay [23, 24]; *ii*) from the privacy perspective, which represents the focus of our work, user inputs are arguably the most critical sources of non-determinism. Our prototype of REAP supports multi-threaded programs (using the Java Pathfinder extension jpf-concurrent [25]) but, at this time, does not handle the reproduction of concurrency bugs. Coping with such kind of bugs would require instrumenting REAP to log, during the symbolic execution phase, any accesses to shared memory, analogously to other sources of non-determinism.

The symbolic execution engine is one of the most crucial components of REAP. REAP uses Java PathFinder [14, 20] (JPF) for this purpose. By default, all variables that are affected by the execution of read calls of the *java.io* library are assumed to be user input and are therefore marked as symbolic. Our anonymization tool is implemented in Java and uses JPF's constraint solving implementation to obtain new input from the path condition. The JPF solving implementation bridges JPF to the actual solver, which can be specified as a parameter. JPF's constraint solving implementation supports several constraint solvers, but in our work we used z3 [26].

## 4 Evaluation

In this section we evaluate REAP's anonymization quality and scalability. REAP was evaluated using six different applications, selected because they manage user sensitive private information, and/or due to their high popularity and to the availability of real bugs. We provide only a brief overview of these applications and of their bugs and references for detailed descriptions.

[4] The REAP prototype is open source: `http://sourceforge.net/projects/fastfixrsm/`

In every plot in this section, the first data point (labeled '-' in the x-axis) represents the results of the OIA phase. Due to the non-deterministic properties of our algorithms, especially of the REAP-BRW, each point of the x-axis of each plot represents the average of 50 runs. The experimental platform used in this study is a machine running the MacOS X Lion operating system, with a 2.5 GHz Intel Core i5 processor and 4 GB of memory. In all experiments, we evaluate each of the algorithms presented in this paper with the test cases presented above for several values of $maxDetourLength$.

### 4.1 Subjects

• *Apache Tomcat* is a large and well-known Java web server (4213 classes, 188 kLOC) that powers numerous large-scale, mission-critical web applications across a diverse range of industries and organizations [27]. In our test case *Tomcat* crashes due to the bug reported in [28]. We aim to anonymize several properties such as security roles, application parameters, amongst many other fields.
• *Apache Xerces* [29] is a popular and large application for parsing and manipulating XML files (1436 classes, 90 kLOC). The bug reported in [30] causes a NullPointerException to be thrown when using external unparsed entities. In our test case *Xerces* parses a xml file that triggers this bug and REAP will attempt to anonymize its content.
• *MySQL/JDBC* [31] is the most popular open source Java database connector (752 classes, 85 kLOC). Our test case is based on the vulnerability reported in [32] and we intend to anonymize the content of the queries.
• *Apache Commons CLI* [33] is a well known application that provides an API for parsing command line options passed to programs (110 classes, 4145 LOC). The bug considered [34] throws an exception when the parser erroneously treats arguments as commands in case of syntax similarities. REAP is intended to anonymize the commands and arguments inserted by the user.
• *PaiNPai* [35] is an personal finances manager (108 classes, 5369 LOC). The bug in this subject is artificial. However, it is a great example of a program that deals with highly sensitive information, such as bank account numbers and other private information of the account holders. Given the confidential nature of such information we consider *PaiNPai* to be an important subject in our evaluation.
• *iDate* is a dating mobile application that finds people matching a specified profile. This application crashes when users use different versions of this application, as they differ in the representation of the input values. In similarity to the *PaiNPai* subject, *iDate* requires the input of very private information. The users devise a personal profile with information such as age, gender, height, weight and also their dating preferences, to be compared with the profiles of other users. We adapted *iDate* (3 classes, 1225 LOC) to run on a desktop computer.

### 4.2 Privacy

We measure privacy using two metrics: leakage, the amount of bits of the original input revealed by the new input, and residue, the amount of bytes of the

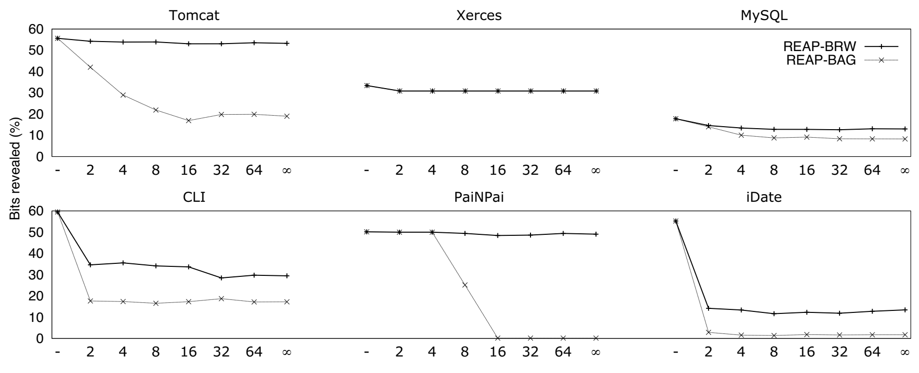

**Fig. 4.** Bar charts showing the leakage

original user input that remain unchanged in the new input. For each test case in every plot, the first point represents the privacy attainable by exploiting the path condition associated with the original execution path $\phi$, generated by the OIA phase (which coincide with that achieved by the solutions in Castro et al. [12] and Clause and Orso [11]). Additionally these plots measure the impact on privacy due to choice of the value of *maxDetourLength*, which we treat as the independent parameter of our study.

**Leakage.** Figure 4 shows the amount of bits revealed in our experiments. The results suggest that even considering a conservative overestimation of the leakage allowed by REAP, evaluated by the path conditions in $\phi \cup \phi'$, REAP-BAG reveals considerably less information than state of the art solutions [11]. In this evaluation, REAP-BAG achieved anonymizations of 83, 22%, on average, and up to 99.84%, whereas REAP-BRW achieved an average of 68.18%. Comparing to the OIA phase, the LM phase of REAP-BAG was able to improve 28.34%, on average, and up to 53.88%, whereas REAP-BRW improved 13.30%. Specifically for each test case, the average improvement of REAP-BAG/REAP-BRW comparing to the OIA phase was respectively: 31.58%/2.07% for *Tomcat*, 2.55%/2.55% for*Xerces*, 8.22%/ 4.61% for *MySQL*, 42.09%/27.26% for *CLI*, 32.19/0.87% for *PaiNPai* and 53.43/42.45% for *iDate*. Figure 4 also suggests that increasing the value of *maxDetourLength* may not provide a path that leaks less information, which was the case specially for *Xerces* and *iDate*. In some cases, REAP-BRW did not show significant improvements when compared to the OIA phase. This is due to its random nature that, in many cases, returns an alternative path that is mostly composed by *narrow* edges. These paths give very few additional solutions and therefore there is little gain in terms of leakage. In *Xerces*, REAP did not anonymize more than 70.15%, as many XML tags need to be fully disclosed if the failure is to be reproduced. Nevertheless these parts are merely XML structural terms and do not reveal sensitive information about the user.

Figure 4 suggests that many of our subjects perform several restrictive logical tests, which force the leakage of significant portion of the user input thereby

```
// each of the following tests fully leak the type of query
if(StringUtils.startsWithIgnoreCaseAndWs(noCommentSql, "INSERT")
 ||StringUtils.startsWithIgnoreCaseAndWs(noCommentSql, "UPDATE")
 ||(...)
```

**Listing 1.1.** MySQL/JDBC

```
//true iff the user
//is a minor
if(IsNomineeMinor){
(...)
}
```

**Listing 1.2.** PaiNPai

```
// if the value of the variable role is in
// securityRoles[], it will be revelealed
for (int i=0; i<securityRoles.length; i++){
  if (role.equals(securityRoles[i]))
    return (true);
```

**Listing 1.3.** Tomcat

**Fig. 5.** Code excerpts exemplifying restrictive logical tests

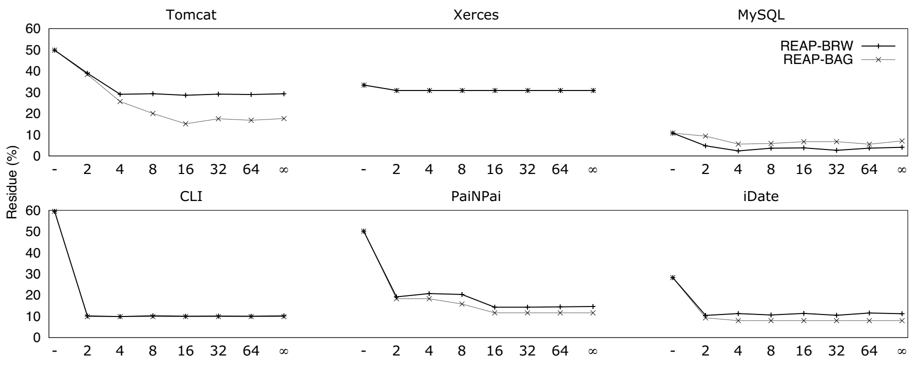

**Fig. 6.** Box plots showing the residue

reducing the effectiveness of an OIA-only approach. Figure 5 presents examples of code excerpts from some of our test cases. In these examples, a mechanism such as the OIA phase (or previous work [11, 12, 18]) would leak all the information introduced by the user. However, REAP may be able to circumvent those branches taken in $\phi$, and find alternative solutions, as suggested in Fig. 4.

**Residue.** Figure 6 presents the residue measurements in our experiments. The results show that REAP is also able to considerably reduce the dissimilarity between the original input and the alternative input. In this evaluation REAP-BAG achieved, on average, residue reductions of 84.93% and REAP-BRW attained 83.07%. Compared with the OIA phase, the LM phase of REAP-BAG improved 23.42% and REAP-BRW improved 21.5%. This means that, before the report is sent, the user is be presented with a very dissimilar input from the one in the original execution.

The main lessons learned in this part of the evaluation are *i*) solutions that consider only the original execution path, such as the OIA phase and mechanisms presented in [11, 12, 18], often leak considerable amounts of information, *ii*) by detouring restrictive logical tests, REAP is able to further anonymize the

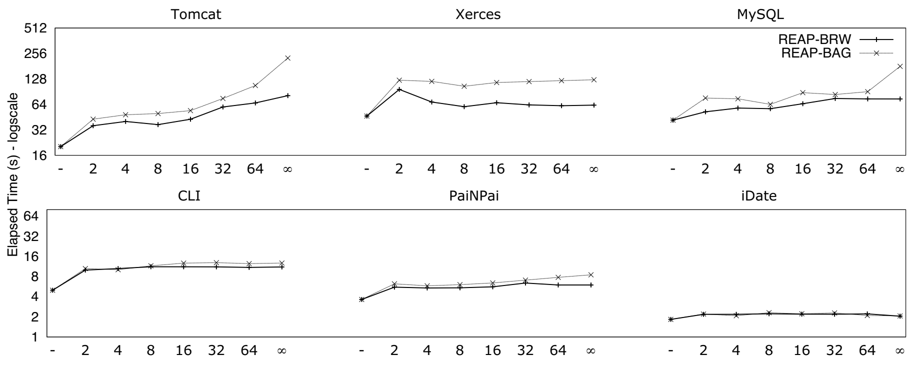

**Fig. 7.** Bar charts showing the execution time

input, *iii*) informed search heuristics, such as REAP-BAG, have the potential to significantly outperform pure random approaches, like REAP-BRW and *iv*) REAP is able to produce alternative inputs that are very dissimilar comparing to the original ones.

## 4.3 Scalability

Figure 7 gives a complete notion of the overhead of REAP when compared with the single phase process of obfuscating using only the original execution path. It is important to note that for each run (with the exception of OIA), the total execution time includes the execution time of the OIA phase and the execution time of the LM phase. The results show that REAP takes at most a few minutes to finish. This is, in practice, perfectly admissible, especially if one considers that REAP will run as a background task executing during idle periods.

Figure 7 suggests that REAP-BRW algorithm is faster than REAP-BAG. This is because, REAP-BRW is biased towards shorter paths. In fact, the number of constraints of the path conditions obtained by REAP-BRW was, on average, 77.23, which is much smaller than the average 223.22 of REAP-BAG and 134.31 of the OIA phase. Additionally REAP performed, on average, 1.09 attempts to reproduce the error. In other words, REAP seldom requires more than one attempt to reproduce the error. In terms of memory usage use REAP-BRW required, on average, 373MB and REAP-BAG 601MB.

We also ran MPP [19] with these subjects and, except for *iDate*, MPP either depleted all available memory or did not find any reproducible alternative path in the first 24 hours of execution. These results confirm what was already found in [19], i.e. it can be prohibitively expensive to compute all possible execution paths of medium/large sized programs, even if this is done offline. For iDate, which is by far the smallest subject in our testbed, the execution time of the MPP Client — even when provided with all pre-computed paths of MPP Server— was two orders of magnitude larger than REAP's (300 sec vs 2 sec).

The main lessons learned in this part of our evaluation are *i*) REAP is a feasible approach for large applications and may not require to be bounded to small values of *maxDetourLength*, *ii*) REAP-BAG is slower than REAP-BRW *iii*) REAP-BRW is likely to find shorter paths and *iv*) MPP is not a feasible approach for medium and large-sized applications.

## 5 Conclusions and Future Work

This paper presented REAP, a system that tackles the issue of user privacy in error reporting. REAP advances the state of the art by increasing privacy through the exploration of alternative execution paths using heuristics that perform bounded deviations in the surroundings of the original path in a scalable way. Our experimental study highlighted that the additional costs, in terms of computation time needed to identify alternative failure-inducing paths, were of at most a few minutes, even for complex applications. Our evaluation also demonstrated that REAP is able to reduce significantly the information leaked with respect to state of the art solutions [11, 12, 18] that do not identify alternative failure-inducing paths, achieving average leakage and residue reductions of 83.22% and 84.93% respectively. We conducted a rigorous analysis of the security properties and guarantees of REAP.

REAP was released as an open-source framework and designed to maximize flexibility and ease of extension. By open sourcing REAP, we hope to foster the interest of other researchers in investigating the design of alternative search algorithms aimed at further enhancing its performance and privacy.

Our future research direction aims at extending REAP in order to support the anonymization of concurrency bugs.

## References

1. Zamfir, C., Candea, G.: Execution synthesis: A technique for automated software debugging. In: EUROSYS, pp. 321–334. ACM, New York (2010)
2. Research Triangle Institute: The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical Report Planning Report 02-3, NIST (2002)
3. Apple Inc: Technical Note TN2123: CrashReporter (2010)
4. Microsoft Corporation: Windows Error Reporting (2012), `http://msdn.microsoft.com/en-us/library/bb513641(VS.85).aspx`
5. Mozilla Foundation: GNOME bug tracking (2013), `http://bugzilla.gnome.org/`
6. Bettenburg, N., Just, S., Schroter, A., Weiss, C., Premraj, R., Zimmermann, T.: What makes a good bug report? In: FSE, pp. 308–318. ACM, New York (2008)
7. Laukkanen, E., Mantyla, M.: Survey reproduction of defect reporting in industrial software development. In: ESEM, pp. 197–206 (2011)
8. Altekar, G., Stoica, I.: Odr: Output-deterministic replay for multicore debugging. In: SOSP, pp. 193–206. ACM, New York (2009)
9. Huang, J., Liu, P., Zhang, C.: Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In: FSE, pp. 207–216. ACM, New York (2010)

10. Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K.H., Lu, S.: Pres: Probabilistic replay with execution sketching on multiprocessors. In: SOSP, pp. 177–192. ACM, New York (2009)
11. Clause, J., Orso, A.: Camouflage: Automated anonymization of field data. In: ICSE, pp. 21–30. ACM, New York (2011)
12. Castro, M., Costa, M., Martin, J.P.: Better bug reporting with better privacy. In: ASPLOS, pp. 319–328. ACM, New York (2008)
13. Wang, R., Wang, X., Li, Z.: Panalyst: Privacy-aware remote error analysis on commodity software. In: Security, pp. 291–306. USENIX, Berkeley (2008)
14. Anand, S., Păsăreanu, C.S., Visser, W.: Jpf-se: A symbolic execution extension to java pathfinder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)
15. Snelting, G.: Combining slicing and constraint solving for validation of measurement software. In: Cousot, R., Schmidt, D.A. (eds.) SAS 1996. LNCS, vol. 1145, pp. 332–348. Springer, Heidelberg (1996)
16. Broadwell, P., Harren, M., Sastry, N.: Scrash: A system for generating secure crash information. In: Security. SSYM 2003, p. 19. USENIX, Berkeley (2003)
17. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE TSE 28(2), 183–200 (2002)
18. Andrica, S., Candea, G.: Mitigating anonymity challenges in automated testing and debugging systems. In: ICAC, pp. 259–264. USENIX, Berkeley (2013)
19. Louro, P., Garcia, J., Romano, P.: Multipathprivacy: Enhanced privacy in fault replication. In: European Dependable Computing Conference, pp. 203–211 (2012)
20. National Aeronautics and Space Administration: Java Pathfinder (2013)
21. Sutton, R.S., Barto, A.G.: Reinforcement learning i: Introduction (1998)
22. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java Optimization Framework. In: CASCON, pp. 125–135 (1999)
23. Machado, N., Romano, P., Rodrigues, L.: Lightweight cooperative logging for fault replication in concurrent programs. In: DSN, pp. 1–12 (2012)
24. VMware: The Amazing VM Record/Replay Feature in VMware Workstation 6 (2011)
25. Ujma, M., Shafiei, N.: jpf-concurrent: An extension of java pathfinder for java.util.concurrent. CoRR abs/1205.0042 (2012)
26. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
27. Apache Foundation: Apache Tomcat (2013), `http://tomcat.apache.org`
28. Apache Foundation: Tomcat Bug Report 29688 (2004), `https://issues.apache.org/bugzilla/show_bug.cgi?id=29688`
29. Apache Foundation: Apache Xerces (2013), `http://xerces.apache.org`
30. Apache Foundation: Xerces Bug Report 4026 (2004), `https://issues.apache.org/bugzilla/show_bug.cgi?id=4026`
31. MySQL: Connector/J (2013), `http://dev.mysql.com/downloads/connector/j/`
32. MySQL: Bug Report 64731 (2012), `http://bugs.mysql.com/bug.php?id=64731`
33. Apache Foundation: CLI (2013), `http://commons.apache.org/cli/`
34. Apache Foundation: CLI bug report CLI-71 (2007), `https://issues.apache.org/jira/browse/CLI-71`
35. Ajey Joshi: PaiNPai (2013), `http://painpai.sourceforge.net/`