Enhancing Privacy Protection in Fault Replication Systems

João Matos, João Garcia and Paolo Romano
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
jmatos, jog, romanop@gsd.inesc-id.pt

***Abstract*—Error reporting systems are valuable mechanisms for enhancing software reliability. Unfortunately, though, conventional error reporting systems are prone to leaking sensitive user information, raising strong privacy concerns.**

In this work we introduce RESPA (*Recursive Shortest Path-based Anonymizer*), a system for generating failure-reproducing, yet anonymized, error reports. RESPA relies on symbolic execution, executed at client side, in order to identify alternative failure-inducing paths in the program's execution graph, and derive the logical conditions, called path conditions, that define the set of user inputs reproducing these executions. Anonymized failure-inducing inputs are then synthesized using any (random) solution satisfying the path conditions.

The search for alternative failure-inducing executions is based on an innovative algorithm that exploits three key ideas: i) RESPA relies on binary search to determine, in an efficient way, which portions of the original execution should be preserved in the alternative one; ii) in order to identify alternative execution paths with low information leakage, RESPA explores the execution graph by leveraging on the Djikstra's shortest path algorithm with information leakage as the distance metric; iii) RESPA ensures provable non-reversibility of the alternative inputs it produces via a recursive technique that anonymizes the alternative inputs found after running the algorithm.

We show via an evaluation based on six large, widely used applications and real bugs that RESPA reduces information leakage up to 99.76%, and on average by 93.92%. This corresponds to an average increase in privacy by 40% with respect to state-of-the-art systems, with gains that extend up to almost 20×.

I. INTRODUCTION

Correcting software errors is a difficult and time-consuming task that represents several billion dollars per year worth of software maintenance costs in Europe and the USA alone [1], [2], [3]. Error reporting systems are valuable mechanisms to enhance software reliability, as they can provide developers with detailed information on the executions that triggered software faults. Such information comes either in the form of memory snapshots capturing the final state of a faulty application, e.g. [4], [5], or in more detailed log traces that capture the sources of non-determinism of the application, e.g. user inputs and thread scheduling [6], [7], [8], [9], [10] that allow to deterministically reproduce the bug at the maintenance site.

Unfortunately, nowadays, security breaches and identity thefts represent serious threats and hence users are likely to take a hard look at conventional types of error reporting mechanism [11]. In fact, whether the user is working on a confidential document or has typed in personal data, sensitive private information is likely to be included either in the memory snapshot taken to generate an error report or in the logs of fault replication mechanisms [12]. Therefore, privacy and security concerns have prevented widespread adoption of many error-reporting systems and ultimately limited their usefulness [13]. Addressing the privacy concerns of error reporting systems is, therefore, a crucial problem for a twofold reason: i) since most computers, nowadays, include error-reporting systems we believe that there is a security risk inherent to these systems that can be exploited to extract confidential information; ii) software maintenance, and consequently its reliability, are degraded if users do not authorize the transmission of error reports, due to privacy concerns.

This work tackles this issue by introducing RESPA, an innovative privacy preserving fault replication system. RESPA relies on a novel technique that explores the graph associated with the different symbolic executions of a program, in order to identify alternative executions that lead to a previously observed failure. RESPA exploits symbolic execution to derive the logical conditions, called path conditions [12], that determine the set of user inputs triggering the alternative failure-inducing executions identified during the exploration phase. An SMT (Satisfiability Modulo Theories) solver is then used to synthesize a random alternative input that satisfies the path condition and is, therefore, guaranteed to reproduce the original bug.

While the idea of anonymizing user inputs in failure-reproduction systems by identifying alternative execution paths has been already explored in recent works [14], [15], the algorithm at the core of RESPA relies on a set of innovative mechanisms that target two important shortcomings of current state-of-the-art systems. Indeed, existing solutions either:
• rely on the exhaustive search of the graph describing all possible (symbolic) executions of a program [14], which are non-scalable and unfit for large applications, or;
• perform bounded explorations attempting *randomized* deviations from the original execution path [15]. While these techniques achieve higher scalability [15] than approaches based on exhaustive searches, as we show in this paper, their effectiveness in identifying alternative inputs can be seriously challenged by bugs whose causes are spread along multiple points of the original failure-inducing execution.

RESPA tackles these issues by introducing an innovative algorithm for exploring the execution graph that combines three key ideas:
• RESPA achieves high scalability by performing best-effort, bounded length explorations in the proximity of the original execution path. Differently from existing systems that rely on randomized techniques [15] to explore the execution graph, RESPA relies on an efficient binary search-based approach

to pinpoint the portions of the original execution path that appear as necessary in order to replay the bug. By taking this information into account, ReSPA avoids fruitless explorations of the search space and focuses on regions that are more likely to yield fruitful results.

• ReSPA searches for alternative paths in the execution graph by focusing on paths that allow for disclosing minimal information on user input. To this end, ReSPA resorts to the Dijkstra's shortest path algorithm [16], using information leakage [17] as the distance metric. This allows ReSPA to exploit the Dijkstra's algorithm to identify the failure-inducing execution path with the lowest leakage among the paths that: $i)$ connect the nodes selected by the binary search phase as necessary for replaying the bug, and $ii)$ are within a bounded radius (using classical distance definition) from the original path. As we will show, the use of this informed search heuristic has a significant impact on the quality of the alternative failure-inducing inputs/execution paths identified by ReSPA, when compared with the ones found by state-of-the-art solutions relying on randomized search heuristics.

• The use of non-randomized search techniques is key to enhance the effectiveness and efficiency of ReSPA. However, the choice of relying on a deterministic exploration phase raises the non-trivial issue of how to ensure the non-reversibility of the algorithm used by ReSPA to identify alternative inputs, i.e., how to avoid that an attacker provided with the alternative inputs produced by ReSPA can trace back the original input (or the original execution path). We solve this problem by applying the ReSPA's anonymization algorithm in a recursive fashion, i.e., by attempting to further improve the anonymization of the alternative inputs generated by previous runs of the algorithm. This mechanism makes the alternative inputs identified by ReSPA and the original user inputs appear equiprobable even to powerful attackers who, given the output produced by ReSPA, can accurately identify the entire set of execution paths that may have been provided as input to ReSPA to generate that output.

We evaluated ReSPA by means of an extensive experimental analysis based on six open-source applications, which include popular large-scale software projects and privacy-sensitive applications. We base the evaluation on real bugs, which are publicly available in the official repositories of these software projects. The results of this study highlight that ReSPA can successfully identify alternative failure-inducing execution paths in scenarios in which state-of-the-art solutions [15] failed. Furthermore, in our experiments, ReSPA achieves an average information leakage of only 6%, improving anonymization with respect to the best state-of-the-art solution by 40% on average and up to 20$\times$ in some applications.

This paper is organized as follows. Section II overviews previous work, Section III presents ReSPA and Section IV presents our experimental results. Finally, Section V concludes the paper.

II. Related work

One of the most popular ways to report errors is through bug-tracking approaches, such as the popular GNOME bug-tracking system [18]. Users can access a project's bug-tracking system, normally via their website, and report a bug. However, three issues affect the effectiveness of bug-tracking systems: $i)$ many users are unwilling to report the bug manually; $ii)$ many users are not aware of what makes a good error report [19]; $iii)$ some users do not know how to replay the bug. Consequently, many bug-tracking sites contain high percentages of reports that the maintenance teams are unable to reproduce.

Initial approaches to automatic error report, such as Windows Error Reporting [4] and Apple Crash reporter [5] involved mainly reporting information collected at the end of a failed program's execution. When an application crashes, the error reporting system gathers information acritically from the state of the process at the moment of the crash and submits it as an error report, if authorized by the user.

Error reports resulting from traditional error reporting systems are very often inadequate to provide developers with sufficient information to reproduce the failure. As stated in plenty of previous work (e.g., [20]) reproducing software bugs is very hard. A traditional error report does not provide any historical information on how the error was reached during the user execution, making the reproduction of the failure a complex and time consuming task for the developers [19], [21], [22]. Further, since error reporting systems do not filter user input, confidential information may end up incorporated in the error report [12]. As already mentioned, this creates severe privacy concerns that hinder the adoption of these tools.

Fault-replication systems tackle precisely this issue, by allowing for the reproduction, at the maintenance site, of a failure observed at the user's machine. In order to replicate the original execution, these mechanisms log any non-deterministic event occurring during an execution, such as user input, network packets and thread interleaving (amongst others). Fault-replication tools provide detailed information on how the error was reached, guiding the programmer (step by step) to the failure. Many fault-replication techniques have been published in past years, e.g., [6], [7], [8], [9], [10]. Unfortunately, since fault-replication systems log all user-inputs to ensure deterministic replay of software faults, they raise even stronger privacy concerns than conventional error-reporting systems (which normally log only a snapshot of the process's state upon the occurrence of a bug).

Intuitively, a direct approach to anonymize error reports is to remove all potentially sensitive information (e.g., Scrash [23]). However, two problems arise from this type of approach. First, we may be amputating the error report from information necessary to locate the causes of the observed failure, making the task of the developers' substantially harder and some times impracticable. Secondly, removing information required to reproduce the observed failure does not necessarily protect the users' privacy, thereby misleading the users to believe that their data is safe when it may not be the case. For example, if we transmit an error report that includes only the stack trace, with no input, we are concealing amongst all inputs that induce that failure, which was the input of the user. However, the provided privacy protection is poor if very few inputs induce such a stack trace. By presenting an incomplete error report to the users, one may delude them to believe that their information is safe, when it may not be the case.

To the best of our knowledge, the idea of anonymizing fault-replication logs by replacing the user's input with an alternative input that induces the same failure was first proposed by Castro et al. [12] and then extended and optimized in various later works [13], [24], [25]. These approaches re-execute the program symbolically through the original execution path that led to the observed failure (i.e., using the original user inputs) and record the sequence of logical constraints imposed by the conditional tests performed on the input (a.k.a., Path Condition [26]). Then, they resort to an SMT solver (e.g., Z3[27]) to obtain an alternative input that satisfies the same sequence of logical constraints.

By estimating the cardinality of the inputs that satisfy the path condition, these approaches can quantify the achieved privacy using classic information theoretic metrics, such as information leakage [12], [17], avoiding the issues that affect solutions [23] that simply remove information from the error report. However, the limitation of these approaches is that the degree of anonymization that they achieve is strictly dependent on the cardinality of the set of inputs that satisfy the path condition of the original execution path. If such path condition contains restrictive logical tests (e.g., testing for equality between a stored password and a string input by the user), these approaches can reveal large portions of the user inputs even though they may be actually unrelated with the bug (e.g., if triggering the bug does not require the user to login).

This limitation has motivated techniques that improve anonymization by searching for alternative executions/path conditions that lead to the same observed failure $\mathcal{F}$. The first work to propose such an approach was MPP [14], which performed symbolic execution through all possible execution paths, thereby obtaining all path conditions that induce $\mathcal{F}$. Clearly, such an approach has strong scalability limitations, as the number of possible execution paths grows exponentially with each logical test performed on the input, which makes it impracticable to use MPP for large, realistic programs.

REAP [15] avoids the scalability issues of MPP by searching for an alternative $\mathcal{F}$-inducing execution path. This search is based on randomized, bounded detours around the original execution path, which is performed until the original fault is replayed. Bounding the scope of the search allows REAP to cope with large programs. However, as we will show shortly, the purely random nature of the exploration performed by REAP can severely limit its ability to identify alternative failure inducing execution paths with non-trivial bugs whose root causes are spread across multiple points of the original execution.

Analogously to REAP, RESPA pursues scalability by bounding the search for alternative execution paths in the proximity of the original one. Unlike REAP, though, RESPA's search phase does not rely on a random strategy — which is unlikely to succeed with complex bugs. Conversely, RESPA uses a deterministic approach that pursues a twofold goal: i) quickly pinpointing, using a binary search approach, which nodes of the original execution path should be preserved in the alternative path, in order to replay the bug; ii) linking these nodes using the Djikstra shortest path algorithm with information leakage as distance metric, in order to promote the exploration of high quality alternative execution paths, i.e., execution paths that minimize information leakage.

III. RESPA

This section is devoted to describe RESPA, and is organized as follows: in Sec. III-A we introduce the notations and assumptions used later on in the paper; in Sec. III-B we overview the proposed system; in Sec. III-C we describe our symbolic execution model and in Secs. III-D and III-E we minutely describe our proposed algorithm.

A. Notations and Assumptions

All notations used in this paper are listed in Appendix A. We denote a software failure with $\mathcal{F}$, which, as in previous work [12], [13], [15], we assume to be observable and uniquely identifiable (e.g., via the stack trace at the occurrence of the bug).

Our goal is to anonymize the original user input $\mathcal{I}$ that triggered $\mathcal{F}$, by replacing it in the error report with an alternative input $\mathcal{I}'$. Additionally we denote with ϕ the original execution path and with $\mathcal{PC}_\phi$ the path condition obtained by re-executing the application symbolically through ϕ. Analogously, we denote with ϕ' an alternative $\mathcal{F}$-inducing path and with $\mathcal{PC}'_\phi$ the respective path condition. We denote with Π the set of all possible inputs (or input domain) and with Π_ϕ the set of all inputs that satisfy $\mathcal{PC}_\phi$, i.e., which replay the execution along the path ϕ.

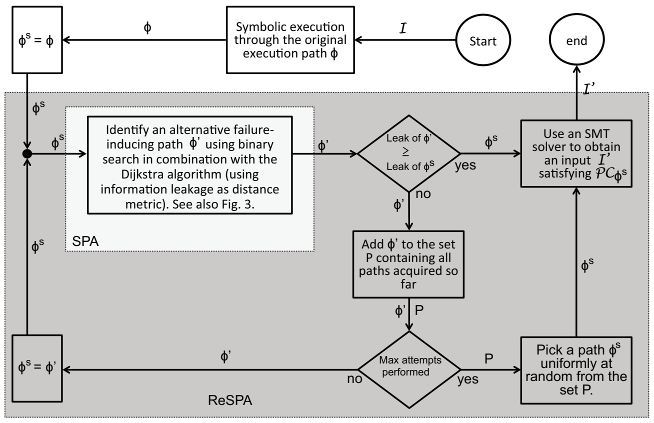
For a matter of simplicity, we treat all user input as potentially sensitive information that can be included in the error report. User inputs include the interactions between the user and the application's user interface, the content of files read during the program's execution and/or any other external data source that interacts with the application and may raise privacy concerns. We are concerned solely with user input, thus we do not consider other types of recording techniques for deterministic replay targeting, e.g., thread scheduling ([7], [8], [9] to name a few), as they are out of the scope of privacy protection. Other sources of program non-determinism besides user input that may influence the reproducibility of the observed failure should be managed using dedicated solutions, which can be employed alongside our system. All deterministic state variables are not anonymized since anyone running the application can anyway determine their values.

Similarly to all previous work in privacy protection in error reporting systems (e.g. [12], [13] to name a few) we assume that all inputs in Π are equiprobable, i.e., we make no assumption on some inputs being more likely than others. This is because we want to keep the mechanisms proposed in this paper application-independent. Removing this assumption would require incorporating application-specific knowledge in order to evaluate the privacy achievable by each target application.

B. Overview

The diagram in Figure 1 provides a high-level overview of the key stages of execution of RESPA.

Compute ϕ and $\mathcal{PC}_\phi$. RESPA begins by taking the input $\mathcal{I}$ logged during the faulty execution and executing the program symbolically using an approach analogous to the one proposed, e.g., by Castro et al. [12]. Via symbolic execution, RESPA



$\phi^s = \phi$
ϕ
Symbolic execution through the original execution path ϕ
I
Start
end
I'
ϕ^s
ϕ^s
Identify an alternative failure-inducing path ϕ' using binary search in combination with the Dijkstra algorithm (using information leakage as distance metric). See also Fig. 3.
ϕ'
Leak of ϕ' ≥ Leak of ϕ^s
ϕ^s
yes
no
Use an SMT solver to obtain an input I' satisfying $\mathcal{PC}_{\phi^s}$
ϕ'
SPA
ϕ^s
Add ϕ' to the set P containing all paths acquired so far
ϕ^s
ϕ' P
$\phi^s = \phi'$
ϕ'
Max attempts performed
no
yes
P
Pick a path ϕ^s uniformly at random from the set P.
ReSPA

Fig. 1: Overview of ReSPA.

identifies the sequence of program statements that compose the original $\mathcal{F}$-inducing execution path, denoted as ϕ, and computes its path condition $\mathcal{PC}_\phi$, i.e., the set of logical constraints on the input domain that cause the program to replay the original $\mathcal{F}$-inducing execution. This phase is described with more detail in Section III-C.

Identify an alternative $\mathcal{F}$-inducing execution path ϕ'. Next, ϕ is fed to as source path, denoted as ϕ^s, to the SPA (Shortest Path-based Anonymization) algorithm. The SPA algorithm aims to identify an alternative $\mathcal{F}$-inducing execution path ϕ', by exploiting two key ideas in synergy: i) binary search is used to determine which nodes of the original execution path ϕ^s should be preserved in ϕ', i.e., which nodes of ϕ^s are suspected to be necessary to reproduce the bug; ii) once binary search fixes the subset of nodes of ϕ^s that should be preserved in ϕ', SPA looks for alternative execution paths that connect such nodes via the Dijkstra shortest-path algorithm. We instantiate the Dijkstra algorithm using information leakage as distance metric, which allows for identifying the execution paths that reveal the minimum amount of information on user input. A detailed description of the SPA algorithm is provided in in Section III-D.

Ensure non-reversibility via recursion. The SPA algorithm is deterministic, i.e., it always provides as output the same alternative path ϕ' when fed with the same source path ϕ^s. In order to prevent the possibility for an attacker to trace back the original execution path ϕ, we execute the SPA algorithm in a recursive fashion, i.e., feeding it with the alternative path ϕ' that it had identified at the end of the previous recursion step. In other words, the SPA procedure is repeated, using the previously computed path ϕ' as the source path ϕ^s. Recursion is performed until either a local minimum in the information leakage of ϕ' is found, or until a maximum number of recursions (whose value is determined randomly) is performed. Details on this aspect of ReSPA are provided in Section III-E.

C. Computing ϕ and $\mathcal{PC}_\phi$

As already mentioned ReSPA uses symbolic execution to identify different execution paths of a program, and which subsets of the input's domain trigger them.

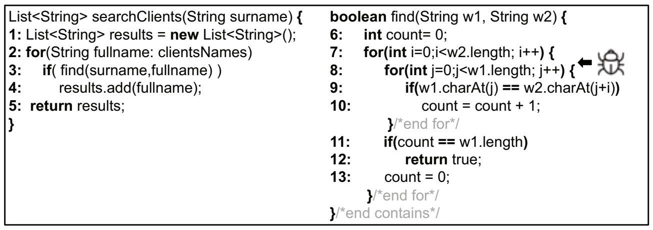
During symbolic execution, when a logical test on the input is performed, two different possible execution states become available, one associated with the case in which the logical test returns true, and another associated with the false case. Each state reached during symbolic execution is associated with a data structure that we call *Node*, which contains the following information (see Algorithm 1): a unique node identifier; a snapshot of the state of the program reached so far during the execution; a list that contains the set of preceding nodes in the current execution path and its path condition; whether the symbolic execution of this node has triggered the occurrence of $\mathcal{F}$; the information leakage [12] associated with the current execution path.

The information leakage definition adopted in ReSPA is an entropy measure based on Shannon's information theory [17] that was proposed originally by Castro et al. [12]. It measures the amount of bits of information on user input revealed by an execution path, or, more precisely, by its path condition. Specifically, denoting with α the fraction of the domain of the application input variables that satisfy a path condition, its leakage is defined as -log$_2(\alpha)$.

In order to derive $\mathcal{PC}_\phi$ we execute the program symbolically from its start, following at each logical test that depends on some input variable, the branch that was taken during the original execution. To this end we use the input logged during

Algorithm 1: The *Node* data structure.

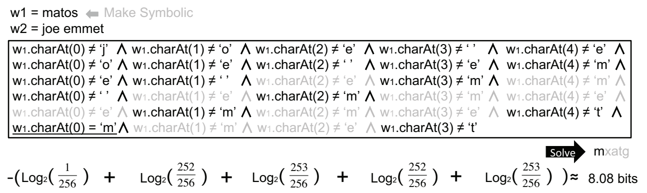
```
struct NODE(
    NODEID ID; //Unique identifier of this node
    State state; //system state at this node
    LinkedList<NODEID > path;//path that lead to this node
    PathCondition $\mathcal{PC} \longleftarrow \emptyset$; //the path condition on how to reach this state
    boolean is$\mathcal{F}$inducing; //does this node induce $\mathcal{F}$?
    float leak; //How many bits are leaked by this PC
)
```



```
List<String> searchClients(String surname) {
1: List<String> results = new List<String>();
2: for(String fullname: clientsNames)
3:     if( find(surname,fullname) )
4:         results.add(fullname);
5: return results;
}
```

```
boolean find(String w1, String w2) {
6:     int count= 0;
7:     for(int i=0;i<w2.length; i++) {
8:         for(int j=0;j<w1.length; j++) {
9:             if(w1.charAt(j) == w2.charAt(j+i))
10:                count = count + 1;
           }/*end for*/
11:        if(count == w1.length)
12:            return true;
13:        count = 0;
       }/*end for*/
}/*end contains*/
```

(a) Buggy code example that tests whether w_2 contains w_1. If w_1 is not contained by w_2 a *StringOutOfBoundsException* can be thrown.



```
w1 = matos
w2 = joe emmet
w1.charAt(0) ≠ 'j'  ∧ w1.charAt(1) ≠ 'o'  ∧ w1.charAt(2) ≠ 'e'  ∧ w1.charAt(3) ≠ ' '  ∧ w1.charAt(4) ≠ 'e'  ∧
w1.charAt(0) ≠ 'o'  ∧ w1.charAt(1) ≠ 'e'  ∧ w1.charAt(2) ≠ ' '  ∧ w1.charAt(3) ≠ 'e'  ∧ w1.charAt(4) ≠ 'm'  ∧
w1.charAt(0) ≠ 'e'  ∧ w1.charAt(1) ≠ ' '  ∧ w1.charAt(2) ≠ 'e'  ∧ w1.charAt(3) ≠ 'm'  ∧ w1.charAt(4) ≠ 'm'  ∧
w1.charAt(0) ≠ ' '  ∧ w1.charAt(1) ≠ 'e'  ∧ w1.charAt(2) ≠ 'm'  ∧ w1.charAt(3) ≠ 'm'  ∧ w1.charAt(4) ≠ 'e'  ∧
w1.charAt(0) ≠ 'e'  ∧ w1.charAt(1) ≠ 'm'  ∧ w1.charAt(2) ≠ 'm'  ∧ w1.charAt(3) ≠ 'e'  ∧ w1.charAt(4) ≠ 't'  ∧
w1.charAt(0) = 'm'  ∧ w1.charAt(1) ≠ 'm'  ∧ w1.charAt(2) ≠ 'e'  ∧ w1.charAt(3) ≠ 't'
mxatg
```

$$-\left(\mathrm{Log}_2\left(\frac{1}{256}\right) + \mathrm{Log}_2\left(\frac{252}{256}\right) + \mathrm{Log}_2\left(\frac{253}{256}\right) + \mathrm{Log}_2\left(\frac{252}{256}\right) + \mathrm{Log}_2\left(\frac{253}{256}\right)\right) \approx 8.08 \text{ bits}$$

(b) The path condition $\mathcal{PC}_\phi$ and the corresponding leakage if the word "matos", provided as input, is compared with "joe emmet", considered as a constant.

Fig. 2: Illustrative example on path condition and leakage computation.

the original execution.

In order to clarify these concepts, which represent the foundation on which we build ReSPA, we report in Figure 2 an illustrative example. The example considers a naive implementation of the longest common subsequence problem, that tests whether a word w_1 is contained by word w_2. The implementation is buggy as it does not enforce the bounds of w_2 when comparing the characters of the two words. For example, if w_1="matos" and w_2="joe emmet", the function *find()* eventually generates an index out of bound exception in line 9, when *i*=*j*=5 and, since w_2 is accessed with offset 10 but it only contains 9 characters.

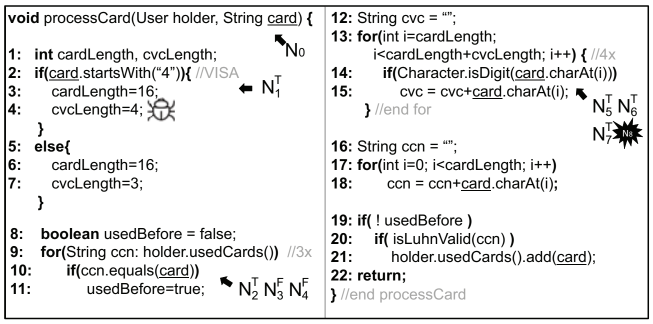
In the example we are assuming that w_1 is a user input, and is as such treated as a symbolic variable. When we execute symbolically the program using the original input (w_1="matos"), we obtain the path condition $\mathcal{PC}_\phi$ exemplified in Fig. 2b. The figure also shows how the leakage of $\mathcal{PC}_\phi$ can be obtained, by computing how many solutions the path condition admits for each byte of the original input.

In the following, we will use the primitive GenerateChildren to abstract over the process of symbolic execution of a node. To this end, we assume that this primitive takes as input a node $\mathcal{N}$, and generates two new nodes $\mathcal{N}^t, \mathcal{N}^f$, which result, respectively, from the symbolic execution of the two outcomes of the logical test (on the input) contained by $\mathcal{N}$. We introduce the InitSymbExec primitive in order to obtain the first node of a program, which stores an empty path condition and the state reached by the program till encountering the first input-dependent logical test. Finally, we use the primitive ϕGet, which takes as input parameter a $\mathcal{F}$-inducing input $\mathcal{I}$, and executes symbolically the program along the original path, returning as output the node in which $\mathcal{F}$ occurred.

D. Shortest Path Anonymizer

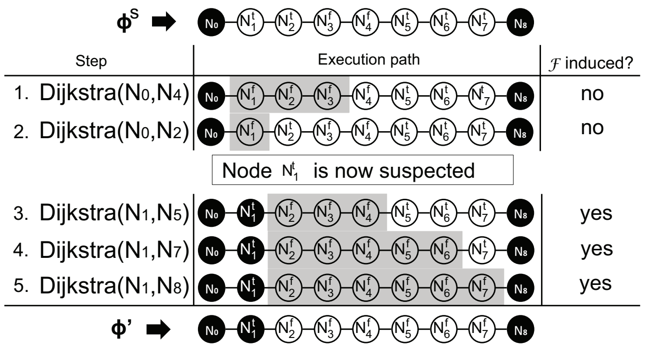
Once the original path ϕ and its path condition are obtained, ReSPA starts invoking the SPA algorithm recursively to obtain alternative $\mathcal{F}$-inducing paths. We will discuss in Sec.III-E why and how ReSPA recursively applies the SPA, but for clarity we first introduce an example that illustrates the functioning of SPA. Next, we present the pseudo-code of SPA.

Example. Figure 3 presents an example designed to help in understanding how SPA works. The code excerpt in Fig. 3a is a function that processes a string, noted card and representing the user input, which contains a credit card number and a CVC number. The invocation of this function is associated with node $\mathcal{N}_0$. The logical test performed in line 2 must return true, for the bug to be reproduced, thereby resulting in node $\mathcal{N}_1^t$. Node $\mathcal{N}_1^t$ contains the bug's root cause, because VISA's CVC numbers are composed of only three digits and, in line 4, the $cvcLength$ variable is assigned with 4. Therefore, the loop at line 15 will iterate 4 times, traversing 4 nodes and crashing in the last one ($\mathcal{N}_9$). Meanwhile, in lines 8-11, a loop checks whether the card has been previously used. In this example we assume the system stores three previously used cards, and that the first of them is equal to the input of the user, thereby



```
void processCard(User holder, String card) {

1:  int cardLength, cvcLength;
2:  if(card.startsWith("4")){ //VISA
3:     cardLength=16;
4:     cvcLength=4;
    }
5:  else{
6:     cardLength=16;
7:     cvcLength=3;
    }

8:   boolean usedBefore = false;
9:   for(String ccn: holder.usedCards())  //3x
10:      if(ccn.equals(card))
11:          usedBefore=true;

12: String cvc = "";
13: for(int i=cardLength;
        i<cardLength+cvcLength; i++) { //4x
14:      if(Character.isDigit(card.charAt(i)))
15:          cvc = cvc+card.charAt(i);
     } //end for

16: String ccn = "";
17: for(int i=0; i<cardLength; i++)
18:      ccn = ccn+card.charAt(i);

19: if( ! usedBefore )
20:    if( isLuhnValid(ccn) )
21:       holder.usedCards().add(card);
22: return;
} //end processCard
```

(a) A buggy code example of a credit card parsing function. If the function is given a string containing a visa credit card number and a three-digit cvc, an exception will be thrown at line 17.



(b) An illustration of SPA's behavior, if ϕ^s is sequence of nodes $\mathcal{N}_0$ to $\mathcal{N}_9$ illustrated in Fig. 3a. Nodes $\mathcal{N}_1$ and $\mathcal{N}_5$ are necessary to reproduce $\mathcal{F}$.

Fig. 3: An example illustrating the SPA algorithm.

traversing nodes $\mathcal{N}_2^t$, $\mathcal{N}_3^f$ and $\mathcal{N}_4^f$. [1]

As already mentioned, SPA relies on a binary search to pinpoint the nodes of the source path ϕ^s that are deemed as necessary to reproduce the bug. In the example in Fig. 3, $\mathcal{N}_1^t$ is necessary to reproduce $\mathcal{F}$, fact that is not known beforehand by the algorithm.

In the first step of the binary search, SPA connects node $\mathcal{N}_0$ to $\mathcal{N}_4^f$ using Dijkstra, using information leakage as a distance metric. This causes $\mathcal{N}_1^t$ and $\mathcal{N}_2^t$ to be replaced, respectively, with $\mathcal{N}_1^f$ and $\mathcal{N}_2^f$ (which yield a lower information leakage).

Next, we continue executing the program complying with the second part of ϕ^s. This is done in order to verify whether $\mathcal{F}$ is still replayed or not. In the example, given that the new sequence of nodes that connects $\mathcal{N}_0$ and $\mathcal{N}_4^f$ does not include $\mathcal{N}_1^t$ (which is necessary), $\mathcal{F}$ is not reproduced. Next, we proceed with the binary search, attempting to connect $\mathcal{N}_0$ with the mid-node between $\mathcal{N}_0$ and $\mathcal{N}_4^f$, i.e., $\mathcal{N}_2^t$. Also in this case, Dijkstra's algorithm will replace $\mathcal{N}_1^t$ with $\mathcal{N}_1^f$, which will lead to a failure in reproducing $\mathcal{F}$. After failing at step 2, SPA considers that $\mathcal{N}_1^t$ is a *suspected* node, therefore SPA decides to include $\mathcal{N}_1^t$

[1]Note that in this scenario, the whole input would be revealed by approaches that rely solely on $\mathcal{PC}_\phi$, e.g., [13], [12]. In fact, since the logical test in line 4 checks for byte-to-byte equality between the entire user input and some concrete string, producing a match in the first iteration, the path condition associated with $\mathcal{N}_2^t$ would leak the entire card string.

in the path ϕ' that it is eventually going to output, and repeat the procedure starting from $\mathcal{N}_1^t$ instead of $\mathcal{N}_0$.

Note that determining whether a node $\mathcal{N} \in \phi$ is necessary to reproduce a failure $\mathcal{F}$ *without any uncertainty* would require identifying all $\mathcal{F}$-inducing execution paths and find $\mathcal{N}_*$ in all of them. Unfortunately, finding all possible execution paths comes at a prohibitive cost for non-trivial programs. In the SPA algorithm, we circumvent this issue by settling for suspicion instead of certainty when it comes to deciding that we cannot replace $\mathcal{N}$. In particular, we *suspect* that $\mathcal{N}$ is necessary to reproduce $\mathcal{F}$, if we exclude $\mathcal{N}$ and only $\mathcal{N}$ from a path known to be $\mathcal{F}$-inducing and $\mathcal{F}$ is no longer reproduced.

Finally, before moving to the detailed description of the algorithm, let us consider steps 3 to 5 illustrated in Fig. 3, which depict the scenario in which, while executing the binary search algorithm, SPA succeeds in replaying $\mathcal{F}$. In step 3 the binary search attempts to connect, with the least leaking path, $\mathcal{N}_1^t$ with $\mathcal{N}_5^t$ and comply with the remaining nodes of ϕ^s. In this case, the bug is replayed despite having replaced $\mathcal{N}_2^t$ (which leaks the entire string) with $\mathcal{N}_2^f$. Hence, the binary search advances rightwards and attempts to connect, using Dijkstra's algorithm, $\mathcal{N}_1^t$ and $\mathcal{N}_7^t$. This leads to replacing $\mathcal{N}_5^t$ with $\mathcal{N}_5^f$ and $\mathcal{N}_6^t$ with $\mathcal{N}_6^f$, which also leads to reproduce $\mathcal{F}$. The procedure ends after inducing $\mathcal{F}$ in step 5, which connects $\mathcal{N}_1^t$ to $\mathcal{N}_8$ with Dijkstra and causes $\mathcal{N}_7^t$ to be replaced with $\mathcal{N}_7^f$.

SPA pseudo-code. The pseudo-code of the SPA algorithm is presented in Alg. 2 and Alg. 3. We start by discussing how SPA searches for alternative paths using the Dijkstra's algorithm (Alg. 2). Next, we present the pseudo code for the binary search algorithm (Alg. 2).

Shortest path based exploration. Let us start by analyzing the functions BDIJKSTRA and ϕCOMPLY in Alg. 2. As the name suggests, the BDIJKSTRA function performs Dijkstra's algorithm starting from the node $next$ and determining the least leaking path to a node having identifier $dstID$. The function also receives a path, called ϕ^s, which represents a reference path used to bound the search space that we explore with the Dijkstra's algorithm. The function returns a node N, whose identifier matches $dstID$ and that will contain information regarding the least leaking execution path connecting $next$ to N.

The BDIJKSTRA function algorithm relies on a Fibonacci heap [28] that prioritizes nodes according to the leakage. However, it bounds the search to a customizable radius $\mathcal{R}$ from ϕ^s in order to control the cost of exploration phase. The *pop* method of the Fibonacci heap (invoked at line 22), removes from the heap and returns the node which has the lowest leakage. Ties are broken in FIFO order. If we attempt to queue a node already present in the heap, the latter is replaced *iff* it has higher leak.

The ϕCOMPLY function executes a portion of the original execution path, passed as input via the ϕ^s parameter, without straying from it. As we have mentioned, the SPA algorithm can execute ϕCOMPLY after having performed some detour from the original path due to the BDIJKSTRA function. Hence, it is neither guaranteed that $\mathcal{F}$ will be reproduced at the last node of ϕ, nor that the last node of ϕ is reachable. The latter is due to the changes performed by BDIJKSTRA that may conflict

Algorithm 2: Pseudo code defining the shortest path based exploration.

```
   Global fields:
1  Failure $\mathcal{F}$; //The observed failure
2  Input $\mathcal{I}$; //The recorded $\mathcal{F}$-inducing input
3  int $\mathcal{R}$; //The radius: $\mathcal{R} \geq 0$
4  FibonacciHeap $< Node > fheap \longleftarrow \emptyset$; //Nodes to explore ordered by
   leakage
5  Set$< Node >$ visited $\longleftarrow \emptyset$; // A cache with visited nodes of $\phi^s$


   BDIJKSTRA (NODE $next$ , NODEID $dstID$ , List<NODEID $>$ $\phi^s$ )
   Returns: a NODE $\mathcal{N}_*$ containing the least leaking path from $next$ to $dstID$
   and such that $\mathcal{N}_*.ID = dstID$. $\mathcal{N}_*$.
   begin
6      if $next.ID = dstID$ then
7          return $next$;
8      if $next.ID \in \phi^s \wedge \nexists \mathcal{N}_* \in$ visited: $\mathcal{N}_*.ID = next.ID$ then
9          add $next$ to visited;
10     NODE $\mathcal{N}_t$,$\mathcal{N}_f$ $\longleftarrow$ GENERATECHILDREN ($next$) ;
11     QUEUE ($\mathcal{N}_t$,$\phi^s$) ; QUEUE ($\mathcal{N}_f$,$\phi^s$) ;
12     return BDIJKSTRA (fheap.pop(),$\phi^s$) ;
   QUEUE (NODE $\mathcal{N}$ , List<NODEID $>$ $\phi^s$ )
   returns: void
   begin
13     if $dist(\mathcal{N}, \phi^s) \leq \mathcal{R} \wedge \mathcal{N}.\mathcal{PC}$ is satisfiable then
14         if $\exists \mathcal{N}_* \in fheap : \mathcal{N}_*.ID = \mathcal{N}.ID$ then
15             if $\mathcal{N}_*.leak > \mathcal{N}.leak$ then
16                 replace $\mathcal{N}_*$ by $\mathcal{N}$ in $fheap$;
17         else
18             push $\mathcal{N}$ into $fheap$;

   $\phi$COMPLY (NODE $next$ , List<NODEID $>$ $\phi^s$ )
   Returns: a NODE $\mathcal{N}_*$ such that $\mathcal{N}_*.ID = \phi^s.tail$
   Pre-condition: $next.ID \in \phi^s$
   begin
19     if $next.ID = \phi^s.tail \vee next.\mathcal{PC}$ is unsatisfiable then
20         return $next$;
21     NODE $\mathcal{N}_t$, $\mathcal{N}_f$ $\longleftarrow$ GENERATECHILDREN ($next$) ;
22     $next \longleftarrow \mathcal{N}_t \in \phi^s \ ? \ \mathcal{N}_t : \mathcal{N}_f$;
23     return $\phi$COMPLY ( next) ;

   HYBRID (NODE $src$ , NUMERIC dstID , List<NODEID $>$ $\phi^s$ )
   Returns: a NODE holding dstID, if $\mathcal{F}$ was induced. Otherwise NODE src.
   begin
24     if $\exists \mathcal{N}_* \in \phi$visited: $\mathcal{N}_*.ID = dstID$ then
25         NODE $\mathcal{N}_{comply} \longleftarrow \mathcal{N}_*$;
       else
26         NODE $\mathcal{N}_{comply} \longleftarrow$ BDIJKSTRA ($src$ , $dstID$ , $\phi^s$) ;
27     NODE $\mathcal{N}_{last} \longleftarrow \phi$COMPLY ($\mathcal{N}_{\text{comply}}$ , $\phi^s$) ;
28     if $\mathcal{N}_{\text{last}}.is\mathcal{F}inducing$ = true then
29         return $\mathcal{N}_{comply}$;
30     return $src$;
```

with ϕ, leading to an unsatisfiable path condition. We deal with these situations simply by breaking the recursive call to ϕCOMPLY and returning the unsatisfiable node when we find it and treat this unsatisfiable execution as not $\mathcal{F}$-inducing.

The HYBRID function jointly uses the BDIJKSTRA and ϕCOMPLY functions to execute symbolically along a path that:

1) starts with the node src;
2) follows the least leaking path from src to a node with identifier $dstId$, which belongs to the path ϕ^s, passed as input parameter;
3) continues executing from $dstId$ to the last node in ϕ^s, which corresponds to the node where $\mathcal{F}$ occurred in ϕ^s.

If $\mathcal{F}$ is reproduced, the HYBRID algorithm returns the destination node of the BDIJKSTRA invocation that is also the parameter of the ϕCOMPLY invocation; if not, the HYBRID algorithm returns the source node that it received as a parameter.

The algorithm starts by checking whether the destination node is present in a list of previously visited nodes of ϕ (ϕvisited, line 24 of the pseudo code). This is an optimization that prevents us from repeating the BDIJKSTRA search to that node. If so, the corresponding system state is restored and the symbolic execution continues through the remaining nodes of ϕ, by invoking the ϕCOMPLY function (line 27). If it is not present in ϕvisited, the BDIJKSTRA algorithm is invoked (line 26) to search for the best path within radius $\mathcal{R}$ that connects the source node and the node with $dstID$ identifier.

The SPA Binary Search. The three functions that were just described are integrated in the SPA Binary Search, formalized in Algorithm 3. The SPA algorithm receives a source path ϕ^s as a parameter and returns a (possibly alternative) failure inducing path ϕ' by means of the inner function _SPA. More precisely, the SPA and _SPA functions return a failure inducing node $\mathcal{N}$, which stores information regarding the execution path that led to $\mathcal{N}$ and the corresponding path condition ϕ' (see Algorithm 1). The _SPA function ï¿$\frac{1}{4}$resorts to several invocations to the HYBRID algorithm in order to identify which nodes of ϕ^s are suspected nodes. It receives the following input parameters: the source execution path ϕ^s, a node $\mathcal{N}_{suc}$ and a node ID $\mathcal{N}_{dst}$, which are the parameter nodes of the HYBRID function and node ID $\mathcal{N}_{fail}$ that is the last $\mathcal{N}_{dst}$ that lead through an execution that did not induce $\mathcal{F}$.

Algorithm 3: Pseudo code defining the SPA algorithm.

```
   SPA (List< NODEID > $\phi^s$ )
   Returns: A $\mathcal{F}$-inducing NODE.
   begin
1      NODE $first \longleftarrow$ INITSYMBEXEC ( ) ;
2      return _SPA ( $\phi^s$ , $first$ , $\phi^s$.get( $\frac{|\phi^s|}{2}$ ) , $\phi^s$.get( $|\phi^s| - 1$ )) ;

   _SPA (List< NODEID > $\phi^s$ , NODE $\mathcal{N}_{suc}$ ,
           NODEID $\mathcal{N}_{dst}$ , NODEID $\mathcal{N}_{fail}$ )
   Returns: A $\mathcal{F}$-inducing NODE.
   begin
3      $\mathcal{N}_{suc} \longleftarrow$ HYBRID ( $\mathcal{N}_{suc}$ , $\mathcal{N}_{dst}$ , $\phi^s$ ) ;

       // Recursion stop condition.
4      if $\mathcal{N}_{suc}$.ID = $\phi^s$.tail then
5          return $\mathcal{N}_{suc}$;

       // Hybrid did not reproduce $\mathcal{F}$.
6      if $\mathcal{N}_{suc}$ did not change then
7          $\mathcal{N}_{fail} \longleftarrow \mathcal{N}_{dst}$;

       //If $\mathcal{N}_{suc}$ and $\mathcal{N}_{fail}$ are subsequent, then $\mathcal{N}_{fail}$ is a suspected node
8      if $\mathcal{N}_{suc}$.ID=$\phi^s$.get($\phi^s$.indexOf($\mathcal{N}_{fail}$)-1) then
9          $\mathcal{N}_{fail} \longleftarrow \phi^s.last$.ID;
10         NODE $\mathcal{N}^t$, $\mathcal{N}^f \longleftarrow$ GENERATECHILDREN ( $\mathcal{N}_{suc}$ ) ;
           //$\mathcal{N}_{fail}$ is either $\mathcal{N}^t$ or $\mathcal{N}^f$. Ensure that we comply with it.
11         $\mathcal{N}_{suc} \longleftarrow \mathcal{N}^t \in \phi^s$ ? $\mathcal{N}^t : \mathcal{N}^f$;
12         $fheap \longleftarrow \emptyset$; visited $\longleftarrow \emptyset$;

13     $\mathcal{N}_{dst} \longleftarrow \phi^s$.get( $\frac{\phi^s.\text{indexOf}(\mathcal{N}_{suc}.\text{ID})+\phi^s.\text{indexOf}(\mathcal{N}_{fail})}{2}$ );

14     return _SPA ( $\phi^s$ , $\mathcal{N}_{suc}$ , $\mathcal{N}_{dst}$ , $\mathcal{N}_{fail}$ ) ;
```

The _SPA algorithm starts by invoking the HYBRID function with $\mathcal{N}_{suc}$ and $\mathcal{N}_{dst}$ received as a parameter (line 1). If the HYBRID algorithm returns the same $\mathcal{N}_{suc}$ received as a parameter, it means that $\mathcal{F}$ was not reproduced. In this case, we assign $\mathcal{N}_{dst}$ to $\mathcal{N}_{fail}$, that is the destination node of the last HYBRID invocation that did not induce $\mathcal{F}$ (lines 6 and 7). Then, in line 13, we set a new destination node id, which is the middle node between $\mathcal{N}_{suc}$ and $\mathcal{N}_{fail}$, and re-invoke the _SPA algorithm at line 14.

From this point on, there are three possible scenarios: $i)$ the invocations to the HYBRID algorithm keep failing, thereby taking $\mathcal{N}_{fail}$ closer to $\mathcal{N}_{suc}$; $ii)$ the invocations to the HYBRID algorithm succeed, which takes $\mathcal{N}_{suc}$ closer to the $\mathcal{N}_{fail}$; $iii)$ a mix between the two, that is, some invocations succeed and some do not. All three scenarios end in the same way: $\mathcal{N}_{suc}$ and $\mathcal{N}_{fail}$ are subsequent nodes of ϕ (line 8). Consequently, SPA considers the node with ID $\mathcal{N}_{fail}$ as a suspected node and since $\mathcal{N}_{suc}$ and $\mathcal{N}_{fail}$ are subsequent, then one of the children nodes of $\mathcal{N}_{suc}$ is $\mathcal{N}_{fail}$, thus SPA complies with it (lines 10-11). Furthermore, _SPA clears the visited nodes set and ϕvisited and restarts the entire procedure starting from the new $\mathcal{N}_{suc}$. Eventually, $\mathcal{N}_{suc}$ keeps moving rightwards in ϕ^s, and the procedure terminates when $\mathcal{N}_{suc}$ is the last node of ϕ^s (lines 4-5).

E. Recursive SPA

In order to guarantee the increased anonymization provided by the larger input domain of the alternative execution path ϕ' output by SPA, it is fundamental to ensure that it is not possible for an attacker to identify the original path ϕ starting from ϕ'. However, given that SPA is deterministic, in a scenario in which ϕ' were only obtainable by SPA if ϕ is the original path, then an attacker may infer ϕ from ϕ' with certainty.

To address the above-mentioned problem, there are two main issues to consider:

- If the path ϕ' achieved by the proposed algorithm leaks a greater or equal amount of bits than the original path ϕ, then we are not interested in using $\mathcal{PC}_{\phi'}$ to obtain $\mathcal{I}'$. Using $\mathcal{PC}_{\phi}$ it is in fact possible to synthesize an alternative/obfuscated input that provides better privacy guarantees.
- If ϕ' leaks less bits than the original path ϕ, then we need to assure that the attacker considers ϕ and ϕ' to have the same probability of being the original path.

To address the second issue, we invoke the SPA algorithm recursively, i.e., starting from $\phi^s = \phi'$ instead of $\phi^s = \phi$. If the new path ϕ'' leaks a larger or equal amount of bits than ϕ', we terminate the procedure. If not, the SPA algorithm is recursively re-invoked until we achieve a path whose leakage is larger or equal than the previous (best) path found. In order to ensure termination, without compromising non-reversibility, we bound the number of recursive invocations to a constant maximum number of attempts, MAX. If we exhaust the budget of available attempts we draw a number i uniformly at random in the range [0, MAX] and return the alternative input produced in the i-th recursion step. This recursive procedure is formalized by the pseudo-code in Algorithm 4.

We analyze the correctness of this approach by assuming the existence of an attacker, who may have access to the error report, the source code of the application and is aware of $\mathcal{F}$ and of how RESPA works, including the value used for the constant MAX. We also assume that the attacker is aware of all the $\mathcal{F}$-inducing path conditions, and that she can "invert" the

ReSPA algorithm, i.e., if provided with the path condition ϕ' output by an execution of ReSPA, the attacker can identify all the path conditions that, when fed as input to ReSPA, produce ϕ'.

We analyze a worst case scenario in which the paths that ReSPA identifies are the only ones that are $\mathcal{F}$-inducing. In fact, the existence of additional $\mathcal{F}$-inducing paths would only create more confusion to the attacker, increasing the complexity of identifying the original path.

We have to distinguish two scenarios: a) the recursion is terminated at lines 8-9, as the leak of the execution path returned by the last recursive call did not decrease, or b) the recursion terminates at lines 12-14 as the leakage monotonically decreased during the last MAX calls to ReSPA. In the latter case it is straightforward to see that the original input/path condition has the same probability of being output as any one of the MAX alternative inputs/path conditions identified by ReSPA. In this case, in fact, the output of ReSPA is selected uniformly at random in this set.

As for case a), the attacker can infer that there is a sequence Σ of path conditions of length at most MAX, denoted as $\phi_*^1, \ldots, \phi_*^i, \ldots, \phi_*^{MAX-1}, \phi_*^{MAX}$ where $\phi_*^{MAX} = \phi'$. The attacker also knows that if the original path condition fed as input to ReSPA was any of the path conditions in Σ, ReSPA would have determined, in all scenarios and with probability 1, ϕ'. Hence, all the path conditions in Σ have the same probability of being the original one.

We note that the presented solution assumes that the attacker has no access to information on the execution speed of the client machine and on the time that ReSPA took to output an alternative input. In fact, in such a case, the attacker could exploit this information to guess how many times ReSPA was executed recursively. On the other hand, this issue could be tackled by using a simple yet effective countermeasure, i.e., injecting random delays between subsequent recursions.

F. Algorithm Analysis

The best case scenario for the SPA algorithm is simple: BDijkstra finds a path connecting the first and last nodes of ϕ and $log_2(|\phi|)$ invocations to the ϕComply algorithm are performed and all reproduce $\mathcal{F}$.

On the other hand, the worst case scenario of the SPA algorithm is when all invocations to the Hybrid algorithm are unsuccessful, and consequently all nodes in ϕ are considered to be required to reproduce $\mathcal{F}$. The proof of the following proposition can be found in Appendix B:

Proposition 1. The SPA algorithm performs in the worst case $log_2(|\phi|!)$ Hybrid invocations.

In terms of nodes visited during the BDijkstra algorithm, the worst case time complexity of the Dijkstra algorithm with a Fibonacci heap is known [28] to be:

$$O(|E| + |V|log(|V|)) \tag{1}$$

where E is the number of edges and V is the number of nodes in the program's execution graph [28] (whose topology is, generally speaking, application-dependent).

Algorithm 4: Pseudo code defining the ReSPA algorithm.

```
   Anonymize ($Input\ \mathcal{I}$, **int** $MAX$)
   **returns** An alternative $\mathcal{F}$-inducing input.
   **begin**
1      Node $\mathcal{N_F} \longleftarrow \phi$Get ($\mathcal{I}$);

2      List<PathCondition> all$\mathcal{PC} \longleftarrow \emptyset$;
3      add $\mathcal{N_F}.\mathcal{PC}$ to all$\mathcal{PC}$;
4      PathCondition $\mathcal{PC} \longleftarrow$ ReSPA ($\mathcal{N_F}.path$, $MAX$, $\mathcal{N_F}.leak$, $all\mathcal{PC}$);

5      Input $\mathcal{I}' \longleftarrow$ ask the solver for an input that satisfies $\mathcal{PC}$;
6      **return** $\mathcal{I}'$;

   ReSPA (List<NodeID > $\phi^s$, **int** max, **float** $leak$, List<PathCondition> all$\mathcal{PC}$)
   **returns** An alternative path condition.
   **begin**
7      Node $\mathcal{N_F} \longleftarrow$ SPA ($\phi^s$);

8      **if** $leak \leq \mathcal{N_F}.leak$ **then**
9          **return** last entry of all$\mathcal{PC}$;
10     add $\mathcal{N_F}.\mathcal{PC}$ to all$\mathcal{PC}$;

11     max $\longleftarrow$ max - 1;
12     **if** max = 0 **then**
13         **int** rand $\longleftarrow$ draw random uniform number [0, $|all\mathcal{PC}| - 1$];
14         **return** all$\mathcal{PC}$.get(rand);

15     $leak \longleftarrow \mathcal{N_F}$.leak;
16     $\phi^s \longleftarrow \mathcal{N_F}$.path;
17     **return** ReSPA ($\phi^s$, max, leak, all$\mathcal{PC}$);
```

IV. Evaluation

We implemented ReSPA in Java and designed it to anonymize the fault-replication logs of Java programs. ReSPA's implementation is open-sourced[2], and relies on a state of the art symbolic execution engine, i.e., Java Pathfinder [29], and SMT solver, i.e., Z3 [27]. The experimental platform used in this study is a machine running a MacOS X Lion operating system, with a 2.5GHz Intel Core i5 processor and 4GB of memory. All presented results are averages across 50 runs. Although ReSPA is deterministic, the SMT solver is not, since the inputs it produces (which, remember, satisfy the path condition output by ReSPA) are randomly generated. As a result, there are slight privacy variations (around 1%) across different runs.

A. Test subjects

We include in our experimental evaluation five well-known applications with real reported bugs and one additional third party application in which we injected a software bug.

We included in our tests Apache Pdfbox [30], which is a library for creating, manipulating and extracting information from/of pdf documents. This is a relevant test in our evaluation because confidential documents are often in the pdf format, thus a substantial amount of information may end up being incorporated in an error report. The bug considered for this benchmark was reported in [31].

Since we compare ReSPA against REAP [15], we included in our evaluation three of the subjects considered in REAP's paper. From the six subjects considered in that paper, we selected: Apache Tomcat [32] with the bug reported in [33],

[2]The prototype implementation of ReSPA is available https://github.com/jgmatos/ReSPA.

Radius ➡	2	4	8	16	32	64	max
⬇ ***Benchmark***				⬇ ***Leak (%) / Iterations***			
Cruise Control	36.6	9.45	5.76	5.29	5.29	5.29	5.29
Pdfbox	8.2	8.2	8.2	8.2	8.2	8.2	8.2
Tomcat	25.1	3.1	1.6	1.6	1.6	1.6	0.24
MySql	5.4	5.4	4.3	3.2	3.2	3.2	3.2
Cli	2.5	2.5	2.5	2.5	2.5	2.5	2.5
VT-Password	40.1	18	17.1	17.1	17.1	17.1	17.1

TABLE I: The table reports the percentage of information leakage for different radius values and benchmarks. For each benchmark, the radius values that minimize leakage are highlighted in green.

Radius ➡	2	4	8	16	32	64	max
⬇ ***Benchmark***				⬇ ***Execution time (s) / Iterations***			
Cruise Control	714.5 / 4	932.0 / 5	1086.2 / 4	565.4 / 3	642.7 / 3	641.8 / 3	645.0/ 3
Pdfbox	293.6 / 4	308.6 / 4	431.9 / 5	291.2 / 4	367.9 / 4	163.1 / 2	159.0 / 2
Tomcat	199.5 / 4	214.8 / 4	236.8 / 4	215.6 / 4	277.4 / 4	186.6 / 3	201.2 / 3
MySql	93.6 / 3	102.6 / 2	102.4 / 2	99.4 / 2	112.8 / 2	101.9 / 2	98.6 / 2
Cli	23.1 / 2	33.5 / 2	46.3 / 2	46.0 / 2	42.6 / 2	41.0 / 2	33.0 / 2
VT-Password	10.6 / 2	13.3 / 2	16.4 / 2	16.1 / 2	16.4 / 2	15.9 / 2	16.1 / 2

TABLE II: The table reports the execution time and total number of iterations performed by ReSPA for different radius values and benchmarks. For each benchmark, the radius values that minimize execution time are highlighted in green.

Mysql [34] with the bug reported in [35] and Apache CLI [36] with the bug reported in [37].

We also use a password validator called VT-Password [38]. This application validates a password, by checking whether it complies with some pre-defined rules, such as minimum/maximum length, containing enough special characters, numbers or uppercase letters, amongst others. Given the highly sensitive nature of the input dealt with by this application, we deem this test subject as particularly interesting privacy-wise. The bug considered was reported in [39].

The application we considered with a non-reported bug is Cruise-control [40]. It is a car engine simulator, that receives commands from the user (e.g. brake, accelerate, amongst others) and simulates the behavior of the engine. The injected bug is triggered by a particular sequence of commands, resulting in an uncaught exception. We considered this benchmark because it generates a very large and complex graph when performing symbolic execution on the input, enabling us to challenge the scalability properties of our algorithms.

In our evaluation we address the following questions:

- Is ReSPA scalable?
- Does the Dijkstra search radius have an impact on the anonymization and efficiency?
- How many invocations to SPA are we required to perform?
- What anonymization gains does ReSPA yield when compared to state-of-the-art solutions?

B. *Anonymization quality and scalability*

We start our study by reporting in Tables I and II experimental data aimed at evaluating the impact of bounding the search radius in RESPA on the resulting efficiency (i.e., execution time and number of performed recursive calls) and anonymization.

By the data in Table I, we can observe a clear, and perhaps unsurprising, trend: in most cases, the larger the radius, the lower (i.e., the better) the information leakage. Indeed, it is expectable that, by extending the search radius, also the chances of identifying paths that substantially increase the domain of known inputs Π_K grow accordingly.

The data in Table II, on the other hand, allows us to evaluate the impact of using larger radius sizes on RESPA's scalability and efficiency. Overall, these data suggest that, independently of the value chosen for the search radius, ReSPA is a practicable solution, especially if we consider that the user is not required to wait for ReSPA to finish, as it is meant to run during idle periods (e.g., overnight). Indeed, on average our experiments required 234 seconds to complete (less than 4 minutes) and the longest test took about 18 minutes. For the experiments with unbounded radius, ReSPA required, on average, 192 seconds (3.2 minutes) to complete.

One may expect that a smaller radius would lead to a lower execution time. However, the results in Table II suggest that this is not often the case. We can observe that a large or unbounded radius, lead to execution times, especially in the Cruise Control and Pdfbox subjects, that are lower than the ones achievable using smaller radius values. On the other hand, we can also observe that, when using lower radius, typically more invocations to the SPA algorithm are required. This explains why their execution time ends up being longer: by setting a smaller radius we are narrowing the search space, thus we are less likely to find the best paths and, as a consequence, we are increasing the room for improvement for the upcoming invocations to the SPA algorithm.

Although it was less frequent, there were some experiments that took longer even with fewer invocations to the SPA algorithm (e.g., Cruise Control with a radius of 8 takes longer than with a radius of 4, with less invocations), which indicates that some invocations may take longer. In fact, the time required for the SPA algorithm to complete depends mostly on the amount of HYBRID invocations which, in turn, increases every time $\mathcal{F}$ is not reproduced.

We can conclude that the amount of time required by RESPA depends mostly on the amount of invocations to the SPA and to the HYBRID algorithms. Also, there is not a clear correlation between the choice of the search radius and RESPA's execution time. In light of these findings we conclude that, at least in the considered testbed, bounding the search radius does not provide any tangible benefit — as, when using an unbounded search radius, we obtained the best anonymization results without necessarily incurring a higher cost in terms of execution time.

C. *Comparison with previous systems*

In Fig. 4 we compare ReSPA with two existing, state of the art, systems: REAP [15] and ϕGET [12]. Indeed, we expect ϕGET to be representative of the anonymization capabilities of similar systems [13], [24], [25], e.g., [13], [24], [25], which share the common idea of relying exclusively on the original path condition $\mathcal{PC}_\phi$ to anonymize user inputs.

Furthermore, given that we concluded in Sec. IV-B that there is no gain in setting a radius, we compare both RESPA

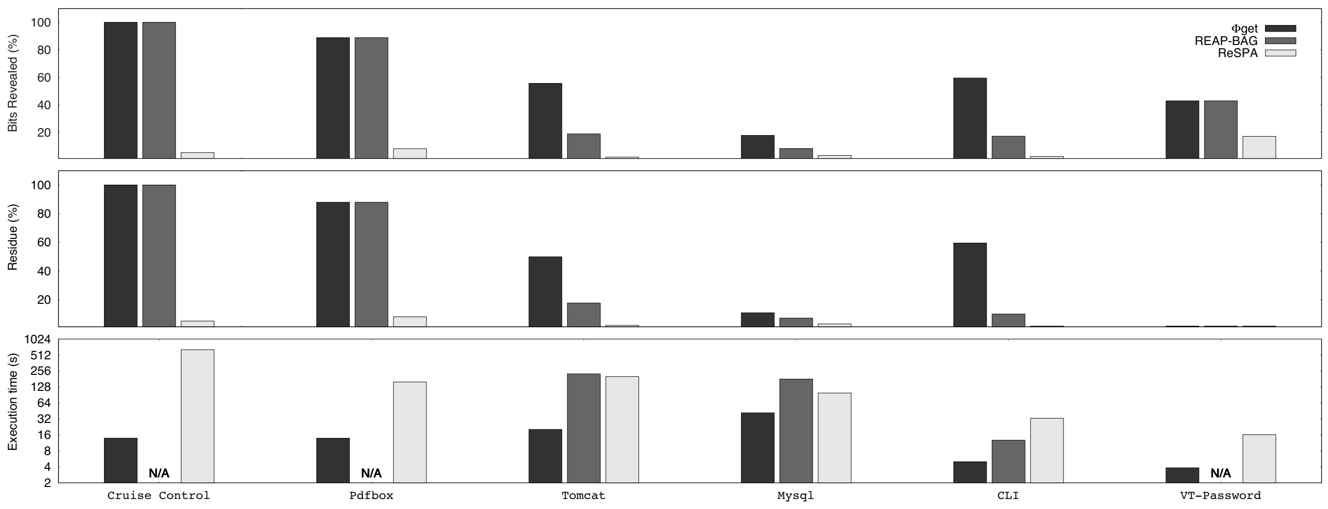


Fig. 4: A comparison between ReSPA, REAP and ϕGet in terms of leakage, residue and execution time, for the experiments with no radius.

and REAP with no radius setting. Additionally, we compare ReSPA against the best configuration of REAP found in that paper: the Bounded Adaptive Greedy (BAG) algorithm.

Fig. 4 includes three plots: *leakage*, *residue* and *execution time*. The *residue* metric was introduced in the work of Clause and Orso [13] and consists of evaluating which percentage of bytes $\mathcal{I}$ and $\mathcal{I}'$ have in common. In other words, this metric tests the similarity between $\mathcal{I}$ and $\mathcal{I}'$.

The top plot highlights that ReSPA reduces leakage up to 99.76%, and on average by 93.92%. This corresponds to an average increase in privacy by 40% with respect to the best performing baseline, i.e., REAP, with gains that extend up to 18.6$\times$ in the case of Cruise Control. Analogous considerations apply if we consider the residue metric.

By analyzing in detail the Cruise Control, Pdfbox and VT-Password test cases, we obtain experimental evidence that confirm our claim that REAP can often fail to find alternative $\mathcal{F}$-inducing paths. In fact, after dozens of attempts, REAP was not able to find an alternative path in these three tests: in such cases, REAP generates an alternative input using $\mathcal{PC}_\phi$, thereby achieving the same anonymization quality of ϕGet. Note that we removed the execution time of REAP for those three cases, as this is directly affected by the budget of attempts allowed to REAP (which is a user-tunable parameter). In those cases, we can observe a very large anonymization difference between ReSPA and previous systems.

Interestingly, even in the test subjects for which REAP does find an alternative path (Tomcat, MySQL and CLI), ReSPA still outperforms REAP significantly. There are two main reasons for this: $i)$ ReSPA identifies alternative paths by means of the Dijkstra's algorithm, which accounts for the cumulated leakage along the entire path. Conversely, REAP-BAG relies solely on local (per node) information. $ii)$ ReSPA's BDijkstra algorithm is guaranteed to eventually find an alternative execution path to some target destination node (i.e., $\mathcal{N}_{dst}$ in Alg. 3). REAP, on the other hand, is more likely to fail identifying alternative paths, since it gives up a "detour" from the original path as soon as its maximum search radius is reached.

V. Conclusions and Future Work

This work proposed a new privacy-enhancing system, called ReSPA, that generates failure-reproducing yet anonymized error reports. ReSPA relies on symbolic execution, executed at client side, to identify *alternative* failure-inducing paths in the program's execution graph and derive the logical conditions that define the set of user inputs that replay these executions.

ReSPA identifies which nodes of the original path are most likely to be required to reproduce the bug, and finds the best path to connect them by instantiating the Dijkstra shortest path algorithm with information leakage as distance metric.

We evaluated ReSPA by means of an extensive experimental study, which includes several real bugs affecting popular applications. Our study shows that ReSPA achieves an average increase in privacy by 40% with respect to state of the art systems, with gains that extend up to approximately 20$\times$.

In our future work, we plan to explore alternative designs for the search heuristic that determines whether a node is to be regarded as necessary to reproduce the bug. Another interesting research avenue is related to how to extend ReSPA in order to relax the assumption (in common to all the approaches in the area we are aware of) on the equiprobability of user inputs.

VI. Acknowledgements

We thank our colleagues from the distributed systems group, as their help and support were essential to accomplish this work. We also thank Professor Alexandre Franciso for the insight on graph algorithms. Furthermore, this work was supported by national funds through Fundação para a Ciência e Tecnologia (FCT), via the projects UID/CEC/50021/2013 and EXPL/EEI-ESS/0361/2013.

References

[1] Nat. Inst. of Standards and Tech.: Software Errors Cost U.S. Economy $59.5 Billion Annually. NIST News Release http://www.nist.gov/director/planning/upload/report02-3.pdf (2002)

[2] Zhivich, M., Cunningham, R.: The real cost of software errors. Security Privacy, IEEE **7**(2) (march-april 2009) 87 –90

[3] Cambridge University: Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year http://www.prweb.com/releases/2013/1/prweb10298185.htm (2013)

[4] Microsoft Corporation: Windows Error Reporting: https://msdn.microsoft.com/en-us/library/bb513641(VS.85).aspx (2015)

[5] Apple Inc.: CrashReporter. http://developer.apple.com/technotes/tn2004/tn2123.html (2010)

[6] Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K.H., Lu, S.: Pres: Probabilistic replay with execution sketching on multiprocessors. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. SOSP '09, New York, NY, USA, ACM (2009) 177–192

[7] Huang, J., Liu, P., Zhang, C.: LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In: FSE. (2010)

[8] Machado, N., Romano, P., Rodrigues, L.: Lightweight cooperative logging for fault replication in concurrent programs. In: Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on. (June 2012) 1–12

[9] Huang, J., Zhang, C., Dolby, J.: Clap: Recording local executions to reproduce concurrency failures. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '13, New York, NY, USA, ACM (2013) 141–152

[10] Machado, N., Lucia, B., Rodrigues, L.: Concurrency debugging with differential schedule projections. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2015, New York, NY, USA, ACM (2015) 586–595

[11] McLaughlin, L.: Automated bug tracking: the promise and the pitfalls. Software, IEEE **21**(1) (jan-feb 2004) 100 – 103

[12] Castro, M., Costa, M., Martin, J.P.: Better bug reporting with better privacy. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XIII, New York, NY, USA, ACM (2008) 319–328

[13] Clause, J., Orso, A.: Camouflage: Automated anonymization of field data. In: Proceedings of the 33rd International Conference on Software Engineering. ICSE '11, New York, NY, USA, ACM (2011) 21–30

[14] Louro, P., Garcia, J., Romano, P.: Multipathprivacy: Enhanced privacy in fault replication. In: Dependable Computing Conference (EDCC), 2012 Ninth European. (May 2012) 203–211

[15] Matos, J., Garcia, J., Romano, P.: Reap: Reporting errors using alternative paths. In Shao, Z., ed.: Programming Languages and Systems. Volume 8410 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2014) 453–472

[16] Dijkstra, E.: A note on two problems in connexion with graphs. Numerische Mathematik **1**(1) (1959) 269–271

[17] Shannon, C.E.: A mathematical theory of communication. SIGMOBILE Mob. Comput. Commun. Rev. **5**(1) (January 2001) 3–55

[18] Mozilla Foundation: GNOME bug tracking system. http://bugzilla.gnome.org/ (2013)

[19] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., Zimmermann, T.: What makes a good bug report? In: FSE. (2008)

[20] Altekar, G., Stoica, I.: Odr: output-deterministic replay for multicore debugging. In: SOSP. (2009)

[21] Laukkanen, E., Mäntylä, M.: Survey reproduction of defect reporting in industrial software development. In: Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on. (Sept 2011) 197–206

[22] Ko, A.J., DeLine, R., Venolia, G.: Information needs in collocated software development teams. In: Proceedings of the 29th International Conference on Software Engineering. ICSE '07, Washington, DC, USA, IEEE Computer Society (2007) 344–353

[23] Broadwell, P., Harren, M., Sastry, N.: Scrash: A system for generating secure crash information. In: Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12. SSYM'03, Berkeley, CA, USA, USENIX Association (2003) 19–19

[24] Wang, R., Wang, X., Li, Z.: Panalyst: Privacy-aware remote error analysis on commodity software. In: Proceedings of the 17th Conference on Security Symposium. SS'08, Berkeley, CA, USA, USENIX Association (2008) 291–306

[25] Andrica, S., Candea, G.: Mitigating anonymity challenges in automated testing and debugging systems. In: Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13), San Jose, CA, USENIX (2013) 259–264

[26] Snelting, G.: Combining slicing and constraint solving for validation of measurement software. In: Proceedings of the Third International Symposium on Static Analysis. SAS '96, London, UK, UK, Springer-Verlag (1996) 332–348

[27] De Moura, L., Bjørner, N.: Z3: an efficient smt solver. In: TACAS. (2008)

[28] Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3) (July 1987) 596–615

[29] Păsăreanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: 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. ISSTA '08, New York, NY, USA, ACM (2008) 15–26

[30] Apache Foundation: Apache Pdfbox. https://pdfbox.apache.org/ (2013)

[31] Apache Foundation: Pdfbox Bug Report 2503. https://issues.apache.org/jira/browse/PDFBOX-2503 (2013)

[32] Apache Foundation: Apache Tomcat. http://tomcat.apache.org (2013)

[33] Apache Foundation: Tomcat Bug Report 29688. https://issues.apache.org/bugzilla/show_bug.cgi?id=29688 (2004)

[34] MySQL: MySQL Connector/J. (2013) http://dev.mysql.com/downloads/connector/j/.

[35] MySQL: MySQL Bug Report 64731. (2012) http://bugs.mysql.com/bug.php?id=64731.

[36] Apache Foundation: CLI. http://commons.apache.org/cli/ (2013)

[37] Apache Foundation: CLI bug report CLI-71. https://issues.apache.org/jira/browse/CLI-71 (2007)

[38] Middleware Services at Virginia Tech: vt-password (2010) https://code.google.com/p/vt-middleware/wiki/vtpassword.

[39] Middleware Services at Virginia Tech: vt-password bug report 207 (2014) https://code.google.com/p/vt-middleware/issues/detail?id=207.

[40] SIR: Software-artifact Infrastructure Repository. http://sir.unl.edu/portal/index.html

Appendix

Notations

Notations used throughout this paper:

$\mathcal{F}$: The observed failure.

Π: The set of all inputs.

$\Pi_{\mathcal{F}}$: The set of all inputs that induce $\mathcal{F}$.

$\mathcal{PC}$: A path condition.

$\mathcal{I}$: The input of the user.

ϕ: The $\mathcal{F}$-inducing execution path that results from $\mathcal{I}$.

$\mathcal{PC}_{\phi}$: The path condition of ϕ.

Π_{ϕ}: The set of all inputs that satisfy $\mathcal{PC}_{\phi}$.

$\mathcal{I}'$: An alternative $\mathcal{F}$-inducing input.

ϕ': The $\mathcal{F}$-inducing execution path that results from $\mathcal{I}'$.

$\mathcal{PC}_{\phi'}$: The path condition of ϕ'.

$\Pi_{\phi'}$: The set of all inputs that satisfy $\mathcal{PC}_{\phi'}$.

Proof of proposition 1

Proof: The worst case of the proposed algorithm consists of the scenario in which all HYBRID invocations fail to reproduce $\mathcal{F}$ and ultimately $\mathcal{I}'$ is generated from $\mathcal{PC}_{\phi}$. In such scenario, the SPA algorithm invokes HYBRID $log_2(|\phi|)$ times without success, which results in suspecting the first node in ϕ to be necessary for replaying the bug. Then this procedure is repeated, excluding the first node of ϕ, thus $log_2(|\phi| - 1)$ times. This is recursively repeated until all nodes are suspected, which results in:

$$log_2(|\phi|) + log_2(|\phi| - 1) + log_2(|\phi| - 2) + \ldots + log_2(1)$$

which can be rewritten as:

$$log_2(|\phi| \times (|\phi| - 1) \times (|\phi| - 2) \times \ldots \times 1)$$

That is equivalent to:

$$log_2(|\phi|!)$$

■