Statistical Symbolic Execution with Informed Sampling

Antonio Filieri
University of Stuttgart
Stuttgart, Germany

Corina S. Păsăreanu
Carnegie Mellon Silicon Valley,
NASA Ames
Moffet Field, CA, USA

Willem Visser and
Jaco Geldenhuys
Stellenbosch University
Stellenbosch, South Africa

ABSTRACT

Symbolic execution techniques have been proposed recently for the probabilistic analysis of programs. These techniques seek to quantify the likelihood of reaching program events of interest, e.g., assert violations. They have many promising applications but have scalability issues due to high computational demand. To address this challenge, we propose a statistical symbolic execution technique that performs Monte Carlo sampling of the symbolic program paths and uses the obtained information for Bayesian estimation and hypothesis testing with respect to the probability of reaching the target events. To speed up the convergence of the statistical analysis, we propose Informed Sampling, an iterative symbolic execution that first explores the paths that have high statistical significance, prunes them from the state space and guides the execution towards less likely paths. The technique combines Bayesian estimation with a partial exact analysis for the pruned paths leading to provably improved convergence of the statistical analysis.

We have implemented statistical symbolic execution with informed sampling in the Symbolic PathFinder tool. We show experimentally that the informed sampling obtains more precise results and converges faster than a purely statistical analysis and may also be more efficient than an exact symbolic analysis. When the latter does not terminate symbolic execution with informed sampling can give meaningful results under the same time and memory limits.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking, Reliability, Statistical methods*

1. INTRODUCTION

Several techniques have been proposed recently for the probabilistic analysis of programs [2, 9, 10]. These techniques have multiple applications, ranging from program understanding and debugging to computing reliability of software operating in uncertain environments.

For example, in previous work [9, 10], we described a bounded symbolic execution of a program that uses a quantification procedure over the collected symbolic constraints to compute the *counts* of the inputs that follow the explored program paths. These counts are then used to compute the probability of executing different paths through the program (or of violating program assertions), under given probabilistic usage profiles. While promising, these exact techniques have scalability issues due to the large number of symbolic paths to be explored.

To address this problem we describe a statistical symbolic execution technique that uses randomized sampling of the symbolic paths. For deciding termination of sampling we investigate two different criteria: Bayesian estimation and hypothesis testing [11, 23]. The first is used to estimate the probability of executing designated program paths while the latter is used to test a given hypothesis about such probability. Unlike in a typical statistical setting where one samples randomly across a concrete input domain, our samples are done in the context of symbolic execution, according to conditional probabilities computed at each branching point in the program. This approach is similar to statistical model checking [33, 35], with the difference that we work with code not with models and we sample symbolic paths, where the probabilistic information is computed based on the collected symbolic constraints.

When using Bayesian estimation, the randomized sampling terminates when pre-specified confidence and error bounds (accuracy) have been achieved. The answer to the analysis problem is not guaranteed to be correct, but the probability of a wrong answer can be made arbitrarily small [35]. However, in practice, the convergence to an answer might be very slow. Hypothesis testing can be faster [35], but both techniques may require a very large number of sample paths to achieve the desired statistical confidence.

To speed up both methods, we propose *Informed Sampling* (IS), an iterative technique combining statistical methods with partial exact analysis. At each iteration, IS randomly samples a set of execution paths and performs a statistical analysis of the sample. The probability of sampling each path is proportional to the number of input points following it under the specified usage profile so not to bias the sample. If the statistical method converged, its result is returned. Otherwise the already sampled paths are pruned out from the execution tree and analyzed exactly. The next iteration will then focus on the analysis of only the remaining part of the execution tree, increasing also the chances of selecting low probability paths that might have not been sampled (and pruned) during the previous iterations.

For pruning the sampling space we propose an efficient procedure that leverages the *counts* of the inputs associated with each explored symbolic path and subtracts them from the counts of all the prefixes along the path. The intuition is that, at the end of each iteration, the counts should keep track of the number of inputs that still need to be explored (sampled) for the execution to follow that path. The counts keep decreasing with each iteration and if a counter be-

comes 0 it means that the sub-tree rooted at that node has been fully explored and can be safely *pruned* from the search space.

For estimating the probability results we propose a combination of exact analysis (for the paths that are pruned in previous iterations) and Bayesian statistical analysis (for the paths sampled in the current iteration over the pruned state space). The analysis terminates when the pre-specified confidence and error bounds have been achieved (for Bayesian estimation) or when the hypothesis is confirmed (for hypothesis testing). The analysis may also terminate when all the paths have been explored, in which case the results will be the same as for the exact analysis. IS converges faster and requires fewer samples than the purely random sampling techniques, since the set of samples is different with each iteration and each pruned path set is analyzed exactly (with confidence 1). Furthermore, the probability of finding the target program events increases with each iteration.

The main focus of this work is on computing non-functional properties of programs, such as the probability of successful termination (or conversely the probability of failure) under a given usage profile [9]. However, statistical symbolic execution with IS can also be used for improving "classical" (non-probabilistic) symbolic execution, in the sense that, if symbolic execution runs out of resources (time, memory) the statistical techniques can be used to provide useful information with statistical guarantees. Note also that the statistical techniques provide an "any time" approach to software analysis: the longer they run, the better the results.

We make the following contributions:

- statistical symbolic execution with two stopping criteria (Bayesian estimation and hypothesis testing) and implementation within the Symbolic PathFinder tool [22];
- IS that converges faster than Monte Carlo sampling;
- an efficient procedure for pruning the state space for incremental symbolic execution;
- combined statistical and exact information for (1) Bayesian estimation and (2) hypothesis testing;
- experimental evidence showing the improvement of IS over state-of-the-art statistical approaches.

2. BACKGROUND

2.1 Symbolic Execution

Symbolic execution is an extension of normal execution in which the semantics of the basic operators of a language is extended to accept symbolic inputs and to produce symbolic formulas as output [15]. The behavior of a program P is determined by the values of its inputs and can be described by means of a *symbolic execution tree* where tree nodes are program states and tree edges are the program transitions as determined by the symbolic execution of program instructions.

The state s of a program is defined by the tuple (IP, V, PC) where IP represents the next instruction to be executed, V is a mapping from each program variable v to its symbolic value (i.e., a symbolic expression in terms of the symbolic inputs), and PC is a *path condition*. PC is a conjunction of constraints over the symbolic inputs that characterizes exactly those inputs that follow the path from the program's initial state to state s.

The current state s and the next instruction IP define the set of transitions from s. Without going into the details of every Java instruction, we informally define these transitions depending on the type of instruction pointed to by IP.

Assignment. The execution of an assignment to variable $v \in V$ leads to a new state where IP is incremented to point to the next instruction and V is updated to map v to its new symbolic value. PC does not change.

Branch. The execution of an *if-then-else* instruction on condition c introduces two new transitions. The first leads to the state s_1 where IP_1 points to the first instruction of the *then* block and the path condition is updated to $PC_1 = PC \wedge c$. The second leads to a state s_2 where IP_2 points to the first instruction of the *else* block and the path condition is updated to $PC_2 = PC \wedge \neg c$. If the path condition associated with a branch is not satisfiable, the new transition and state are not added to the symbolic execution tree.

Loop. A *while* loop is unrolled until its condition evaluates to false or a pre-specified exploration depth limit is reached. Analogous transformations are applied to other loop constructs.

The initial state of a program is $s_0 = (IP_0, V_0, PC_0)$, where IP_0 points to the first instruction of the main method, V_0 maps the arguments of main (if any) to fresh symbolic values, and $PC_0 = true$. A program may also have one or more terminal states that represent conditions such as the successful termination of the program or an uncaught exception that aborts the program execution abruptly.

Although our approach can be customized for any symbolic execution system, we focus on Symbolic PathFinder (SPF) [22] that works at the Java bytecode level.

2.2 Probability Theory

The possible outcomes of an experiment are called *elementary events*. For example, the rolling of a 6-sided die may produce the elementary events 1, 2, 3, 4, 5, and 6. Elementary events have to be *atomic*, i.e., the occurrence of one of them excludes the occurrence of any other. The set of all elementary events is called a *sample space*. In this paper, we consider only finite and countable sample spaces, meaning that the underlying set of elementary events is countable and finite.

DEFINITION 1 (PROBABILITY DISTRIBUTION). *Let Ω be the sample space of an experiment. A* probability distribution *on Ω is any function $Pr : \mathscr{P}(\Omega) \to [0,1] \subset \mathbb{R}$ that satisfies the following conditions (probability axioms):*

- $Pr(\{x\}) \geq 0$ *for every elementary event x*
- $Pr(\Omega) = 1$
- $Pr(A \cup B) = Pr(A) + Pr(B)$ *for all events A, B with $A \cap B = \emptyset$*

The pair (Ω, Pr) constitutes a probability space.

DEFINITION 2 (CONDITIONAL PROBABILITY). *Let (Ω, Pr) be a probability space. Let A and B be events $(A, B \subseteq \Omega)$, and let $Pr(B) \neq 0$. The* conditional probability *of the event A given that the event B occurs, is:*

$$Pr(A|B) = \frac{Pr(A \cap B)}{Pr(B)}$$

$Pr(A|B)$ is also referred to as the probability of A given B.

DEFINITION 3 (LAW OF TOTAL PROBABILITY). *Let (Ω, Pr) be a probability space and $\{B_n : n = 1, 2, 3, \dots\}$ be a finite partition of Ω. Then, for any event A:*

$$Pr(A) = \sum_n Pr(A|B_i) \cdot Pr(B_i)$$

The law of total probability can also be stated for conditional probabilities:

$$Pr(A|X) = \sum_n Pr(A|X \cap B_i) \cdot Pr(B_i|X)$$

where B_i are defined as in Definition 3 and X does not invalidate the assumptions of Definition 2.

2.3 Probabilistic Analysis

We follow previous work [9] where we defined a symbolic execution framework for computing the probability of successful termination (and alternatively the probability of failure) for a Java software component placed in a stochastic environment. A failure can be any reachable error, such as a failed assertion or an uncaught exception. For simplicity, we assume the satisfaction of target program properties to be characterized by the occurrence of a target event, but our work generalizes to bounded LTL properties [35].

To deal with loops, we run SPF using *bounded* symbolic execution, i.e., a bound is set for the exploration depths. The result of symbolic execution is then a finite set of paths, each with a path condition. Some of these paths lead to failure, some of them to success (termination without failure) and some of them lead to neither success nor failure (they were interrupted because of the bounded exploration) – the latter are called *grey* paths.

The path conditions produced by SPF consequently form three sets: $PC^s = \{PC^s_1, PC^s_2, \ldots, PC^s_m\}$, $PC^f = \{PC^f_1, PC^f_2, \ldots, PC^f_p\}$ and $PC^g = \{PC^g_1, PC^g_2, \ldots, PC^g_q\}$, according to whether they lead to success, failure, or were truncated. Note that the path conditions are disjoint and cover the whole input domain. In other words, the three sets form a complete partition of the input domain [15, 22].

Not all input values are equally likely, and we employ a *usage profile* to characterize the interaction of the software and the environment. It maps each valid combination of inputs to the probability with which it may occur. In [9] we provide an extensive treatment of usage profiles and how they are used for the probabilistic computations. For simplicity, and without loss of generality, we will assume here that the usage profile is embedded in the code. This can be done with every usage profile where the probabilities p_i are described by arbitrary precision, rational numbers. More general usage profiles, such as Markov Chains, can be embedded as well; they are analyzed in a bounded way.

Given the output of SPF, and assuming the constraints from the usage profile have been embedded in the code, the probability of success is defined as the probability of executing program P with an input satisfying any of the successful path conditions (recall the path conditions are disjoint):

$$Pr^s(P) = \sum_i Pr(PC^s_i) \tag{1}$$

The failure probability $Pr^f(P)$ and "grey" probability $Pr^g(P)$ have analogous definitions; it is straightforward to prove that $Pr^s(P) + Pr^f(P) + Pr^g(P) = 1$. $Pr^g(P)$ can be used to quantify the *impact* of the execution bound on the quality of the analysis ($1 - Pr^g(P)$).

In this paper we focus on sequential programs with integer inputs. In other work we provide treatment of multi-threading [18], input data structures [9], and floating-point inputs [2] (see Section 7).

2.3.1 Quantification Procedure

We compute the probabilities of path conditions using a quantification procedure (e.g., [6, 9, 10]) for the path conditions. We use LattE [6] to count models for linear integer constraints but our work generalizes straightforwardly to other tools such as QCoral [2] (for arbitrary floating point constraints) and Korat [3] (for heap data structures; see [9] for details).

Given a finite integer domain D, model counting allows us to compute the number of elements of D that satisfy a given constraint c; we denote this number by $\sharp(c)$ (a finite non-negative integer). By definition [20], $Pr(c)$ is $\sharp(c)/\sharp(D)$ (where $\sharp(D)$ is the size of the domain implicitly assumed to be greater than zero).

The success probability (or failure or grey probability) can then be computed using model counting as follows.

$$Pr^s(P) = \sum_i Pr(PC^s_i) = \frac{\sum_i \sharp(PC^s_i)}{\sharp(D)} \tag{2}$$

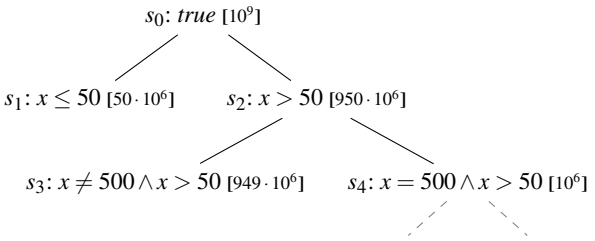

Figure 1: Partial symbolic execution tree of the example.

3. EXAMPLE

In this section we illustrate the proposed techniques with a simple example. Consider the code in Listing 1. Assume the goal is to estimate the success probability of the method *test*, i.e., the probability of not reaching line 6, where an exception would be raised. Assume the input variables x, y, and z range over the integer domain [1..1000]. The size of the input domain is $10^3 \cdot 10^3 \cdot 10^3 = 10^9$ points. In practice the domains can be much larger. Note that the size of the domain does not affect the complexity of the counting procedure, which only depends on the number of input variables and the number of constraints [6, 28].

Listing 1: Illustrative example

```
1  void test(int x, int y, int z) {
2     if(x<=50) {
3        // Do some work
4     } else {
5        if(x==500 && y==500 && z==500){
6           assert false;
7        }
8        // Do more work
9     }
10 }
```

Assuming a uniform usage profile, the probability of hitting the failure (*assert false*) is 10^{-9}, since there is only one point in the input domain that can lead to failure.

To illustrate how sampling works, consider Figure 1, where a part of the symbolic execution tree of Listing 1 is reported. For each branching point (represented as a node) we show both the path condition and the corresponding counter in square brackets. These counters are initially computed by LattE as the paths are explored, and stored for re-use. The first time a branch is encountered, the counters are used to compute the probability of each alternative, and a randomized choice is made accordingly (see Section 4). For example, the probability of moving from s_2 to s_3 is $949/950$; the number of points satisfying *PC* at s_3 is $949 \cdot 10^6$ while the number of those satisfying *PC* at s_4 is 10^6, which together sum up to the number of points in *PC* at the parent state s_2. In our approach, a second simulation run would reuse this computation, making repeated sampling efficient.

Statistical symbolic execution with IS starts the first iteration by performing a small number of samples, as dictated by the probabilities of the branching conditions. Assume for simplicity that at the end of the first iteration, only the path $s_0 \rightarrow s_2 \rightarrow s_3$ has been taken (perhaps multiple times); this is reasonable since the transitions along this path have significantly higher probabilities than their peers. The counter for the final *PC* along this path is $949 \cdot 10^6$. This number is then subtracted from all the counters upward along

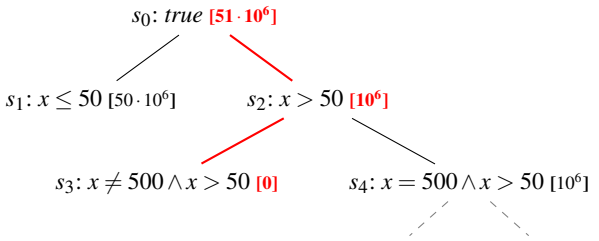

Figure 2: Symbolic execution tree: counters updated after one iteration

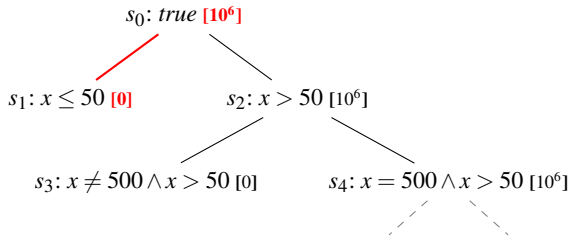

Figure 3: Symbolic execution tree: counters updated after two iterations; only one path left to explore.

the path, yielding the updated counters in Figure 2. A new iteration begins, where the sampling is now guided by the updated values of the *PC* counters. Note that in this iteration, the transition from s_2 to s_3 can never be taken, since its counter is 0. At the same time notice that the probability of following the path leading to the subtree containing the error (rooted at s_4) has increased from $1/10^3$ in the first iteration to $1/51$ in the second iteration.

Assuming the more likely path $s_0 \rightarrow s_1$ is sampled in the second iteration, the counters are updated according to the numbers shown in Figure 3. In this last iteration, the only remaining path $s_0 \rightarrow s_2 \rightarrow s_4$ is taken, leading to the exploration of the subtree containing the assert violation. Monte Carlo sampling without pruning would miss the path leading to the violation, unless an infeasibly large number of samples were taken. For example, after 20000 samples the error is still undetected.

After each iteration we also *combine* the information obtained from an exact analysis of pruned paths and a Bayesian inference over sampled paths (over the pruned state space) to determine if enough evidence was collected about the probabilities of the events of interest. For simplicity we omit the details for this example but we will describe this at length later in the paper.

4. STATISTICAL SYMBOLIC EXECUTION

We first describe Statistical Symbolic Execution, which computes an approximate solution to the probability of success (or failure) of a program, based on *sampling* carefully chosen program paths. Informed Sampling will be described in the next section.

The basic idea is to address the probability computation as a statistical inference problem. First, a randomized sampling procedure generates a finite number of simulation runs and classifies each of them as either satisfying or violating a given property ϕ (e.g., an assertion in the code). Second, suitable statistical methods are applied to either *estimate* the probability of ϕ from an analysis of the samples or to *test a hypothesis* about this probability.

Similar techniques have already been explored in the literature on statistical model checking [33, 35], which typically phrase the statistical inference problem in the context of formal verification of probabilistic models, i.e., transition systems annotated with probabilities such as Markov Chains or Markov Decision Processes.

We describe here how we adapted Bayesian statistical techniques [35] in the context of symbolic execution of Java programs, where no model is assumed and the probabilistic information is derived via model counting over the symbolic constraints (in combination with the usage profile).

4.1 Monte Carlo Sampling of Symbolic Paths

Typically, a Monte Carlo method defines the solution of the problem as the parameter of a hypothetical population, and then generates a random set of samples from which statistical estimates of this parameter can be obtained [12, 24].

In the context of symbolic execution, we define a sample as the simulation of one symbolic path. Whenever a branch is encountered during such simulation, the decision to proceed along either of the alternative branches has to be taken according to the probability of satisfying the corresponding branch conditions under the current usage profile.

Every time a condition is encountered, the simulation has to decide whether to follow the true or the false branch. In particular let PC_{branch} be the path condition at the current state, and let c be the branching condition at that state. The path condition after taking the "then" branch is $PC_{then} = c \wedge PC_{branch}$ while the path condition after taking the "else" branch is $PC_{else} = \neg c \wedge PC_{branch}$.

Similar to [10] we associate to each of PC_{branch}, PC_{then}, and PC_{else} a counter of the number of points in the input domain satisfying the path condition, identified by $C(PC_{branch})$, $C(PC_{then})$ and $C(PC_{else})$, respectively. The first time a path condition PC is encountered, its counter is initialized through model counting to the number of points of the input domain that satisfies PC: $C_i = \sharp(PC)$. After its initialization, the value of each counter is stored and reused through the simulation process.

We can compute the branch probabilities as follows:

$$p_{then} = \frac{\sharp(\, c \wedge PC_{branch})}{\sharp(\, PC_{branch}\,)} = \frac{C(PC_{then})}{C(PC_{branch})}$$
$$p_{else} = \frac{\sharp(\, \neg c \wedge PC_{branch})}{\sharp(\, PC_{branch}\,)} = \frac{C(PC_{then})}{C(PC_{branch})} \quad (3)$$

From Equation (3) it is straightforward to note that $\sharp(PC_{then}) + \sharp(PC_{else}) = \sharp(PC_{branch})$ and $p_{then} + p_{else} = 1$.

The decision whether to take the *then* or the *else* branch can now be taken randomly according to their respective probabilities p_{then} and p_{else}. It remains to show that making the sampling choices locally at each branch is equivalent to making the choices over the complete PC, i.e., we do not introduce any statistical bias. This is implied by the following result:

THEOREM 1. *For a path with $PC = c_1 \wedge c_2 \wedge \ldots \wedge c_n$ and the branching conditions encountered in the given order, the path probability given by Pr(PC) is equal to the product of the conditional probabilities at each branch given by $Pr(c_1 \mid true) \times Pr(c_2 \mid c_1) \times Pr(c_3 \mid c_2 \wedge c1) \times \ldots \times Pr(c_n \mid c_{n-1} \wedge \ldots \wedge c_1)$.*

PROOF. *From Section 2.3.1 we have that $Pr(PC) = \frac{\sharp(PC)}{\sharp(D)}$ where D is the complete finite domain and from Equation 3 we can rewrite the product of conditional probabilities as*

$$\frac{\sharp(c_1)}{\sharp(D)} \times \frac{\sharp(c_1 \wedge c_2)}{\sharp(c_1)} \times \frac{\sharp(c_1 \wedge c_2 \wedge c_3)}{\sharp(c_1 \wedge c_2)} \times \ldots \times \frac{\sharp(c_1 \wedge \ldots \wedge c_n)}{\sharp(c_1 \wedge \ldots \wedge c_{n-1})}$$

which is equal to $\frac{\sharp(c_1 \wedge \ldots \wedge c_n)}{\sharp(D)} = Pr(PC)$. □

4.2 Bayesian Inference and Stopping Criteria

The samples generated by the Monte Carlo simulation described in the previous section need to be analyzed to estimate the probability μ of the program to satisfy a given property ϕ. Bayesian statistical techniques exploit Bayes' theorem to update the *prior* information on the probability μ after every observed sample. The prior is a probability distribution that summarizes all the available information (including its lack) gathered through sampling [7, 17].

As explained in [35], a prior for μ can be formalized via the Beta distribution $\mathscr{B}(\alpha,\beta)$ (see details in [11, 23, 35]). By setting α and β it is possible to specify the initial assumption about μ as follows. Assume the software engineer has an initial guess $\tilde{\mu}$ about μ, for example based on the analysis of previous versions of the software or on the quality of third-party components involved. One way of encoding such knowledge as a prior distribution is:

$$\begin{cases} \alpha &= \tilde{\mu} \cdot N_p \\ \beta &= (1-\tilde{\mu}) \cdot N_p \end{cases} \tag{4}$$

where $N_p \geq 1$ represents the "trust" on $\tilde{\mu}$ as if it was observed on N_p samples. If no initial information is available, a "non-informative" prior can be used, such as $\mathscr{B}(1/2,1/2)$ [23]. The meaning is that we give the same chance $(1/2)$ to both possible outcomes, and we give small trust to it. We treat grey paths either *optimistically* or *pessimistically*, meaning that they are considered as either success or failure, as desired by the user.

When new samples are gathered, they are used to update the prior, leading to the construction of the *posterior* distribution. In particular, if n samples have been collected with n_s of them satisfying ϕ, the parameters α' and β' of the posterior distribution will be computed as:

$$\begin{cases} \alpha' &= \alpha + n_s \\ \beta' &= \beta + n - n_s \end{cases} \tag{5}$$

This information can then be used for statistical estimation or hypothesis testing as explained below.

4.2.1 Bayesian Estimation

We use Bayesian estimation [11, 23] to compute a value that is close to μ with high probability. More precisely, we compute an estimate $\hat{\mu}_B$ such that:

$$Pr(\hat{\mu}_B - \varepsilon \leq \mu \leq \hat{\mu}_B + \varepsilon) \geq \delta \tag{6}$$

where $\varepsilon > 0$ is the *accuracy* and $0 < \delta < 1$ is the *confidence*; the accuracy determines how close the estimate has to be to the real unknown μ and the confidence expresses how much this result can be trusted [35].

Recalling that the posterior has a Beta distribution, with parameters α' and β', Equation (6) can be restated as:

$$F_{\mathscr{B}(\alpha',\beta')}(\hat{\mu}_B + \varepsilon) - F_{\mathscr{B}(\alpha',\beta')}(\hat{\mu}_B - \varepsilon) \geq \delta \tag{7}$$

where $F_{\mathscr{B}(\alpha',\beta')}(\cdot)$ is the cumulative distribution function of the posterior distribution, i.e., it computes the probability for a random variable distributed according to the posterior to assume a value less than or equal to the argument [20].

From the correctness of Bayesian estimation [11, 23] (i.e., it always converges to the real value of μ after enough samples are collected), Equation (7) can be used as a sequential stopping criterion to decide how many samples are needed to achieve the accuracy and confidence goals.

If the estimation converges with the prescribed accuracy and confidence, the estimate $\hat{\mu}_B$ is defined as the expected value of the posterior distribution:

$$\hat{\mu}_B = \frac{\alpha'}{\alpha' + \beta'} \tag{8}$$

An estimate on the number of samples that is required to achieve the accuracy and confidence goals is discussed in [35]. In general, this number is highly sensitive to the accuracy parameter, while increasing the prescribed confidence has a lower impact on the number of samples.

4.2.2 Bayesian Hypothesis Testing

We use hypothesis testing as an alternative stopping criterion for termination. Hypothesis Testing [20, 23] is a statistical method for deciding, with enough confidence, whether the unknown probability μ is greater than a given threshold θ ($H_0 : \mu \geq \theta$). Alternatively, we may want to evaluate the complementary hypothesis $H_1 : \mu < \theta$.

Similar to estimation, hypothesis testing starts from prior knowledge and updates it with the information obtained through sampling until enough evidence is provided in support of either H_0 or H_1. The procedure aims at estimating the odds for hypothesis H_0 versus H_1, which can be computed as follows [35]:

$$\frac{Pr(H_0|S)}{Pr(H_1|S)} = \frac{Pr(S|H_0)}{Pr(S|H_1)} \cdot \frac{Pr(H_0)}{Pr(H_1)} \tag{9}$$

where S is the set of samples collected, and $Pr(H_0)$ and $Pr(H_1)$ are the probability of the hypothesis to be true given the prior knowledge, respectively; $Pr(H_0) = 1 - F_{\mathscr{B}(\alpha,\beta)}(\theta)$ and $Pr(H_1) = 1 - Pr(H_0)$.

The ratio $Pr(S|H_0)/Pr(S|H_1)$ is called a *Bayes factor* and can be used as a measure of relative confidence in H_0 versus H_1 [23, 35], i.e., it quantifies how many times H_0 is more likely to be true than H_1 given the evidence collected through sampling. The values $Pr(H_0|S)$ and $Pr(H_1|S)$ represent the probability of the two hypotheses to be true after samples S have been collected. Since all the information gathered from the samples is embedded in the posterior distribution, the latter is used to compute $Pr(H_0|S) = 1 - F_{\mathscr{B}(\alpha',\beta')}(\theta)$ and $Pr(H_1|S) = 1 - Pr(H_0|S)$. Thus, the Bayes factor **B** corresponding to the posterior odds for hypothesis H_0 can be computed from Equation (9) after some algebraic simplifications as:

$$\mathbf{B} = \frac{Pr(S|H_0)}{Pr(S|H_1)} = \frac{Pr(H_1)}{Pr(H_0)} \cdot \left(\frac{1}{F_{\mathscr{B}(\alpha',\beta')}(\theta)} - 1 \right) \tag{10}$$

If no preference among the two hypotheses is provided by the prior, e.g., when a non-informative prior is used, the initial value of the ratio $Pr(H_1)/Pr(H_0)$ is 1.

Equation (10) can be used to define a sequential stopping criterion. In particular, sampling can stop when the odds in favor of one of the hypotheses (the Bayes factor **B**) is greater than a given threshold T, i.e., when a relative confidence of at least T is obtained from data to support one of the hypotheses. A precise quantification of the number of samples needed to achieve convergence for Bayesian hypothesis testing is discussed in [31, 35]. In general, hypothesis testing is faster than estimation, although its performance degrades when θ is close to the (unknown) probability μ [31].

5. INFORMED SAMPLING

A weakness of statistical analysis is the large number of paths that may need to be explored and the slow convergence to a result, within the desired confidence. To address this problem, we introduce here *Informed Sampling* (IS), an iterative technique that combines Monte Carlo sampling with *pruning* of already explored

Algorithm 1: Statistical Symbolic Execution with Informed Sampling

```
1  exploredD ← 0;
2  successD ← 0;
3  repeat
4      numSamples ← 0;
5      numSuccess ← 0;
6      successPCs ← {};
7      exploredPCs ← {};
8      repeat
9          π ← MonteCarloSample();
10         let PC be the path condition of path π;
11         numSamples ← numSamples + 1;
12         exploredPCs ← exploredPCs ∪ {PC};
13         if π ⊨ φ then
14             numSuccess ← numSuccess + 1;
15             successPCs ← successPCs ∪ {PC};
16         end
17         updatePrior();
18     until StopCombinedEst() ∨ numSamples ≥ N_I;
19     exploredD ← exploredD + ♯(exploredPCs);
20     successD ← successD + ♯(successPCs);
21     if StopCombinedEst() then
22         return;
23     end
24     pruneOutPaths(exploredPCs);
25 until exploredD = domainSize;
26 return;
```

paths. Furthermore, to obtain a precise estimation of the probability of satisfying property ϕ, IS combines information from two sources: the first is based on the exact probabilistic analysis (described in Section 2) for the pruned paths and the second is based on Bayesian inference (as described in Section 4.2) for the sampled paths. We describe IS in more detail below.

5.1 Algorithm

Symbolic execution with Informed Sampling is described at a high level by Algorithm 1. Assume for simplicity that we are interested in the success probability of the program with respect to a property ϕ (the algorithm can also be applied to failure probability with only minor modifications). N_I is a pre-specified number of samples per iteration. Assume also that we treat the grey paths optimistically.

The algorithm works through a number of iterations (lines 3–25). At each iteration, IS first tries to tackle the verification problem through Bayesian inference. For this task, it takes a pre-specified number of Monte Carlo samples (lines 8–18) as dictated by the conditional probabilities computed from the code. At each iteration, the algorithm keeps track of the following values:

- *numSamples* counts the number of sampled paths
- *numSuccess* counts the number of sampled paths that lead to success
- *exploredPCs* stores the *PC*s of explored paths
- *successPCs* stores the *PC*s of explored paths that lead to success

The algorithm also computes *exploredD* and *successD* which keep count of total explored inputs and explored inputs that lead to success. These values are computed using model counting (lines 19–20) and are used in the combined estimators as described below.

As before we use as stopping criteria for sampling either Bayesian estimation or Hypothesis testing (high-level procedure *StopCombinedEst()* in lines 18 and 21). However for IS we use combined estimators that enhance the Bayesian inference with precise information obtained from symbolic paths. If the (combined) Bayesian estimator converges to the desired confidence or if the (combined) estimated probability satisfies the hypothesis, this result is reported and the analysis stops. The iterative process can also terminate when the whole domain was analyzed (line 25).

After each iteration the symbolic paths explored so far are pruned out of the execution tree (line 24) and analyzed using the exact method (Section 2); the results are used to build the combined estimator. This improves the efficiency of the inference procedure because it accounts for all the information obtainable from the path conditions of explored paths. Indeed, each sampled path has a path condition which is used in the exact analysis to quantify how many input values from the domain will follow the execution along that path. For example, referring to Figure 1, the symbolic path $s_0 \rightarrow s_2 \rightarrow s_3$ accounts for more concrete program paths than the path $s_0 \rightarrow s_1$; however this information is ignored by the purely statistical inference, which treats symbolic paths as concrete paths.

Pruning using Counters.

Recall that for each path condition *PC* we maintain a counter $C(PC)$ to count the number of solutions. Initially these counters are computed using off-the-shelf quantification procedures such as LattE. At each iteration, IS performs sampling, as guided by the *PC* counters (see Section 4.1). For each sampled (non-duplicate) path, with final *PC* counter n, IS updates all the counters for the prefixes of *PC* along the path (to the root of the symbolic execution tree) by subtracting n, and a new iteration starts (with the updated counters). Thus, for each pruned *PC* only a small number of arithmetic operations is required, with no significant impact on the overall computation time.

At the end of each iteration, the counters keep track of the number of inputs that need to be sampled to follow that path. If a counter becomes 0 it means that the sub-tree rooted at that node has been fully explored, and it does not need to be sampled again. Therefore we can safely *prune* it from the search space. If the counter of the root node becomes 0 the analysis stops, because the whole domain was analyzed exactly.

After each pruning, exact information is obtained for a fraction of the input domain. This fraction needs no longer to be considered for statistical inference, allowing the latter to focus on the remaining part of the domain. Furthermore, the overall confidence in the result grows, since there is no uncertainty about the fraction of the domain analyzed exactly.

Estimation with IS.

The combined estimator, denoted here as $\hat{\mu}$, is defined through the mixture of an exact estimator, denoted μ_E, and a Bayesian estimator, denoted $\hat{\mu}_B$. E refers to the inputs that follow the paths explored in previous iterations of IS (and can therefore be analyzed Exactly), while B refers to the inputs that have not been explored yet (and therefore can only be used in Bayesian estimation). A hat ("ˆ") denotes an approximate value.

For the input points that have already been explored, we can compute the exact probability μ_E. Recall that *successD* denotes the number of input points corresponding to the pruned successful paths and *exploredD* is the total number of points corresponding to pruned paths. Then:

$$\mu_E = \frac{successD}{exploredD} \tag{11}$$

and or the rest of the input domain we have at each iteration just the Bayesian estimator:

$$\hat{\mu}_B = \frac{numSuccess + \alpha}{numSamples + \alpha + \beta} \tag{12}$$

where for both α and β we use $1/2$ as default. By the law of total probability (Definition 3) we can combine the exact and Bayesian estimators:

$$\hat{\mu} = (1 - f_E) \cdot \hat{\mu}_B + f_E \cdot \mu_E \tag{13}$$

where f_E is the fraction of the domain that has been pruned out up to the previous iteration, i.e., $exploredD/\sharp(D)$. The number of samples to take at each iteration is decided according to a sequential stopping criteria, and it is bounded by the maximum value N_I provided by the user.

Hypothesis Testing with IS.

For hypothesis testing recall that we base the decision on the posterior odds of the hypothesis $H_0 : Pr(P \models \phi) \geq \theta$ versus $H_1 : Pr(P \models \phi) < \theta$. For IS we compute the posterior odds based on a combined estimator similar to the one described in Equation (13):

$$\hat{\mu}^{H_0} = (1 - f_E) \cdot \hat{\mu}_B^{H_0} + f_E \cdot \mu_E^{H_0} \tag{14}$$

where $\hat{\mu}_B^{H_0}$ is the Bayesian posterior estimator defined in Section 4.2.2 for the probability $Pr(H_0|S)$. S is the set of samples taken during the current iteration (i.e., $1 - F_{\mathscr{B}(\alpha',\beta')}(\theta)$, where $\alpha' = \alpha + numSuccess$ and $\beta' = \beta + numSamples - numSuccess$ are the parameters of the posterior Beta distribution of the Bayesian estimator). $\mu_E^{H_0}$ is equal to 1 if the result μ_E of the partial exact analysis is greater than or equal to θ, and equal to 0 otherwise; f_E is the fraction of the domain that has been pruned out up to the previous iteration, as described in the previous section.

Early Termination.

We further enhance the IS procedure to check for additional sufficient termination conditions determined by the partial exact analysis of pruned paths. Indeed, the actual value of μ is by definition in the interval:

$$\frac{successD}{\sharp(D)} \leq \mu \leq 1 - \frac{failD}{\sharp(D)} \tag{15}$$

where $failD = exploredD - successD$.

We use these lower and upper bounds to test against the hypothesis and decide early termination of the IS procedure. Indeed, if $successD/\sharp(D) \geq \theta$ the hypothesis is necessarily true; while if $1 - failD/\sharp(D) < \theta$ the hypothesis is necessarily false. In both cases we stop the iterative process and return the result to the user. This check is performed in *StopCombinedEst()*.

5.2 Discussion

Combined Estimators are Unbiased and Consistent.

The construction of the combined estimator of Equation (13) is an application of stratified sampling, where the population (the input domain) is partitioned into disjoint subsets to be analyzed independently; the local results are then linearly composed, assigning each one a weight proportional to the size of the corresponding subset [5]. An estimator obtained through stratified sampling is unbiased (i.e., its expected value converges to the measure it estimates) and consistent (i.e., its variance converges to 0 when the number of samples goes to ∞) if the local estimators used for each subset of the partition are unbiased and consistent [5].

For the portion of the domain analyzed exhaustively, μ_E is by definition the actual measure it estimates. Thus it is trivially unbiased and consistent (indeed the variance of a number is always zero). For the portion of the domain subject to statistical estimation, we adopt the standard Bayesian estimator for the parameter of a Bernoulli distribution. Proofs that it is unbiased and consistent can be found, for example, in [11, 23, 35]. Thus, the combined estimator is in turn unbiased and consistent.

Termination.

If IS explores the whole domain, that is $exploredD = \sharp(D)$, the process terminates with the same results as for the exact analysis. Since at each iteration the number of samples to collect for Bayesian inference is greater than zero, IS is guaranteed to terminate, in the worst case, when the whole domain has been analyzed exactly. (Note that we assumed the domain is finite.)

Faster Convergence for Bayesian Estimation.

A benefit of mixing the Bayesian estimator $\hat{\mu}_B$ with μ_E is a faster convergence to the criterion of Equation (6). Indeed, if an input falls in the portion of the domain analyzed exactly, our estimate is perfectly accurate (with confidence 1) by definition. Otherwise it will provide confidence $\hat{\delta}$:

$$\hat{\delta} = (1 - f_E) \cdot \delta_B + f_E \tag{16}$$

Thus, to meet the prescribed confidence δ as a whole, the Bayesian estimator is required to just satisfy the relaxed confidence δ_B.

$$\delta_B \geq \frac{\delta - f_E}{1 - f_E} \tag{17}$$

During the first iteration, when $f_E = 0$, δ_B needs to satisfy the original convergence criterion of Bayesian estimation (i.e., the prescribed δ). However, with each iteration, f_E increases, thus relaxing the constraint on δ_B.

Faster Convergence for Hypothesis Testing.

As for Bayesian hypothesis testing, the process terminates as soon as the odds in favor of H_0 overcome those in favor of H_1 by a factor T decided by the user. To understand the benefit in terms of the convergence rate provided by the IS estimator of Equation (14), we need to consider the ratio of the posterior odds $Pr(H_0|S)/Pr(H_1|S)$. If H_0 is actually true, $Pr(H_0|S)$ will converge to 1 (and consequently $Pr(H_1|S)$ to 0) the more samples are collected. The other way around, if H_0 is false $Pr(H_0|S)$ will converge to 0 (and $Pr(H_1|S)$ to 1). The convergence of the estimator $\hat{\mu}^{H_0}$ can be evaluated again considering f_E. Since after each iteration f_E grows, the room for the uncertainty derived from the use of Bayesian estimation is always bounded by a decreasing factor $1 - f_E$. The more execution paths are pruned out and analyzed exactly, the more such uncertainty is reduced, usually speeding up the convergence of the combined estimator.

Detecting Errors with Random Exploration.

The iterative pruning of the input domain increases the chances of random exploration to detect errors. To show this, let us consider an error path with path condition PC^R. Let B^i represent the set of the paths targeted by random sampling during iteration i, and let B^{i+1} represent the set of paths targeted by sampling during iteration $i+1$. If the error path is not detected at iteration i, we will show that the probability of catching PC^R is higher at iteration $i+1$.

Let us assume, for simplicity, that only one path is sampled per iteration (the worst case for our proof). The probability of sampling

PC^R at iteration i is $Pr(PC_R|B^i)$. If it is sampled, then the error has been detected. Otherwise a sampled path with condition PC^i is removed from B^i. Since $B^{i+1} = B^i - PC^i$ it follows that at iteration $i+1$, the probability $Pr(PC^R|B^{i+1})$ of catching PC^R is higher than in the previous iteration:

$$Pr(PC^R|B^i) = Pr(PC^R|PC^i) \cdot Pr(PC^i) + Pr(PC^R|B^{i+1}) \cdot Pr(B^{i+1}) \quad (18)$$

Note that $Pr(PC^R|PC^i) = 0$ because we assumed that the sampled path with PC^i was not the error path with PC^R, it follows that:

$$Pr(PC^R|B^{i+1}) = \frac{Pr(PC^R|B^i)}{Pr(B^{i+1})} \quad (19)$$

Again, assuming that PC^R has not been detected yet, necessarily $B^{i+1} \neq \emptyset$ and thus $Pr(B^{i+1}) > 0$. The example in Section 3 illustrates this phenomenon: the error is very hard to detect with purely random exploration but it can be easily detected with IS.

Number of Samples; Incremental Symbolic Execution.

The maximum number of samples to take in each iteration of IS allows us to select different operation modes for the algorithm. If a very large number of samples are allowed during each iteration, IS reduces to Bayesian inference as described in Section 4.2. On the other hand, if $N_I = 1$ the impact of the Bayesian estimation becomes negligible, since it will almost surely not converge after a single sample, making IS perform an *incremental* exact analysis by selecting, pruning, and analyzing one symbolic path per iteration.

Thus IS can be used to improve on "classical" symbolic execution by providing for a new kind of *incremental analysis* where the next path to be analyzed is selected according to the Monte Carlo Sampling described in Section 4.1. In this way IS will likely cover the most probable paths first, computing also the fraction of the domain these paths cover. This results in an "any time" approach where it is possible to interrupt the execution when enough of the input domain has been covered, even if the analysis cannot be exhaustively completed within a reasonable time.

Values of N_I between the two extremes trade off the effort Bayesian estimation is allowed to take to converge during a single iteration with the number of iterations required to converge. Choosing a good value for N_I depends on the specific problem. We will discuss its choice for several applications in Section 6. Another option is to "adapt" the value of N_I with the number of iterations, e.g., by starting small to quickly prune out paths with high likelihood of execution and gradually increasing the value of N_I to stress-test the parts of the state space that have a small likelihood of execution.

False Positives or Negatives.

Statistical hypothesis testing, being a randomized procedure operating on a limited number of samples, may produce false negatives or positives, i.e., it may reject a hypothesis that is actually true and vice versa. This problem can occur especially when the analyzed programs are very large and the probability of success or failure is close to the extremes (0 or 1) [34]. In the next section we show an instance of the problem. For Bayesian hypothesis testing, it has been proved that the probability of obtaining spurious results is bounded by $1/T$ [35], where T is the threshold set by the user (see Section 4.2.2).

For IS, pruning reduces the possibility of spurious results since it limits the possibility of wrong conclusions to the fraction of the domain analyzed with the Bayesian estimator $(1 - f_E)$. Also note that the sufficient conditions that we added to IS, for early termination with hypothesis testing, do not suffer from incorrect results because they rely on exact methods. Thus they improve the quality of the overall approach since if IS terminates due to the sufficient conditions, its results are always correct.

6. EXPERIENCE

We implemented the statistical symbolic execution techniques described in this paper in the context of SPF [22], an open-source toolset. We plan to make our tool available for download. Sampling is parallelized using a map-reduce algorithm. Path counters are shared and reused in subsequent sampling phases.

In this section we compare IS with both an exhaustive analysis and a purely statistical approach. We report on the analysis of the following software artifacts:

OAE: the Onboard Abort Executive (OAE) [21] software component manages the Crew Exploration Vehicle's ascent abort handling developed at NASA. OAE has 1400 LOC, 36 input variables ranging over large domains, and fairly complicated logic encoding the flight rules (a path condition can have approx. 60 constraints). We are interested in the probability of the OAE not raising a mission abort command.

MER: models a component of the flight software for JPL's Mars Exploration Rovers (MER) [1]; it consists of a resource arbiter and two user components competing for five resources. MER has 4697 LOC (including the Polyglot framework). The software has an error (see [1]) and is driven by input test sequences. We analyze two versions: *MER (small)* for sequence length 8 and *MER (large)* for sequence length 20; the latter cannot be analyzed fully with symbolic execution because of the huge number of execution paths.

Sorting: an implementation of Insertion sort. We calculate the probability of sorting an array of size n in exactly $n(n-1)/2$ comparisons, i.e., the worst case. A large number of paths need to be analyzed ($n!$), but only 1 path leads to the worst case. Despite being a simple algorithm, this example is very challenging for statistical techniques due to the low probability of hitting any failure. We analyze a version for $n = 7$.

Windy: a standard example in the reinforcement learning community that involves a robot moving in a grid from a start to a goal state. A crosswind can blow the robot off course and an added weight to the robot counter-balances that. We analyze two versions: *Windy (small)* has a 5×4 grid and solutions limited to 5 moves, and *Windy (large)* has a 9×4 grid and 12 moves. The latter cannot be analyzed exhaustively with symbolic execution because of the very large number of paths the robot may follow. We consider reaching the goal state in the specified number of moves as a success.

OAE was analyzed on a Red Hat Linux 64bit machine with 4Gb of memory and a 2.8GHz Intel i7 CPU. The other software was analyzed on an Ubuntu Server 12.04.4 LTS 64bit with 16Gb of memory and a 3.10GHz quad-core Intel Xeon CPU E31220.

Estimation. Table 1 shows some of our results for the probability estimation problem. δ and ε represent the target confidence and accuracy, N_I is the number of samples per iterations, *Iter* is the number of iterations completed during analysis, *Estimate* is the result computed, and *Time* is time consumption in milliseconds. For all the examples in this table we assume a uniform usage profile for the inputs and we treat grey paths optimistically.

$\delta = 1$ denotes that IS has been used for incremental exact analysis (thus computing the actual success probability without uncertainty), while $N_I = 100000$ means that the analysis was purely statistical (no IS). There are several observations to make about these numbers:

- For OAE, ε and N_I do not seem to play a role in the number of iterations required, or the time consumption. This is because after the first iteration, even with 100 samples, more than 99.8% of the domain is pruned out, and IS achieves the required confidence

Table 1: Estimation results (* means non-convergence, ** means exhaustive analysis)

	δ	ε	N_I	Iter	Estimate	Time
OAE	1	–	1	3754	0.999999981808025	629,818
	0.99	10^{-2}	100	2	0.9998659113123208	42,110
	0.99	10^{-2}	1000	1	0.9998659113123208	40,326
	0.99	10^{-3}	100	2	0.9998659113123208	42,223
	0.99	10^{-3}	1000	2	0.9998659113123208	540,678
	0.99	10^{-3}	100000	–	0.9995836802664446	317,074
	0.99	10^{-5}	100000	–	0.999990000199996*	31,654,165
	δ	ε	N_I	Iter	Estimate	Time
MER (small)	1	-	1	122	.75	100,420
	0.99	10^{-3}	100	9	0.75**	168,414
	0.99	10^{-3}	1000	9	0.7499661471528664	210,635
	0.975	10^{-5}	100	9	0.7499263416861861	167,695
	0.975	10^{-5}	1000	9	0.7499828915718787	211,332
	0.99	10^{-5}	100	9	0.7499254209572634	166,871
	0.99	10^{-5}	1000	9	0.7499686952166291	210,464
	0.99	10^{-3}	100000	–	0.749705005899882*	25,784,373
	0.99	10^{-5}	100000	–	0.7510049799004019*	25,803,456
	δ	ε	N_I	Iter	Estimate	Time
Sorting	1	–	1	5040	0.999988	946,681
	0.99	10^{-3}	100	68	0.9999069235294118	1,943,537
	0.99	10^{-3}	1000	18	0.9999636105960265	1,823,969
	0.99	10^{-5}	100	69	0.9999527117647059	2,689,889
	0.99	10^{-5}	1000	18	0.9999856615894039	2,195,849
	0.99	10^{-3}	100000	-	0.9995836802664446	307,192
	0.99	10^{-5}	100000	-	0.999990000199996*	9,113,719
	δ	ε	N_I	Iter	Estimate	Time
Windy (small)	1	–	1	614	0.004073625	70,554
	0.99	10^{-3}	100	16	0.004164252291666667	7,348
	0.99	10^{-3}	1000	11	0.004073625	8,275
	0.99	10^{-5}	100	17	0.004100003958333333	123,204
	0.99	10^{-5}	1000	11	0.004073625**	148,843
	0.99	10^{-3}	100000	–	0.00438745663560302	1,859,271
	0.99	10^{-5}	100000	–	0.004309913801723965*	6,319,183

quickly. Indeed, OAE has a few "success behavior" paths accounting for most of the executions, while the abort paths share a small probability of being taken under the uniform profile (we will later report on a different mission profile). Thanks to Monte Carlo sampling, the former are very likely to be sampled, and then pruned, first. IS is dramatically faster than the purely statistical approach, and its estimate is also closer to the true value.

- MER (small) has several execution paths occurring for roughly the same number of inputs. IS needs more iterations to prune them out and achieve the high accuracy and confidence goals (unlike in the OAE case). However, due to the small number of paths (122), after 9 iterations at least 99% of the domain is covered, pushing the convergence of the IS estimator, which outperforms the Bayesian estimator. Notice also that the latter does not reach the required confidence for such high accuracy: after 100000 samples it reaches a confidence of only .5346 and .0058, for ε equal to 10^{-3} and 10^{-5}, respectively.
- For Sorting, only N_I significantly influences the number of iterations. This reflects the fact that – initially – the 5040 paths are equally likely, and so we expect the impact of pruning in IS to be small. This scenario is particularly suitable to Bayesian estimation, which converges for $\varepsilon = 10^{-3}$ after about 2500 samples. Since we limited N_I to smaller values, IS was not able to achieve convergence by its statistical component until pruning covered a large portion of the domain. When the accuracy is raised to $\varepsilon = 10^{-5}$, the Bayesian estimator is not able to converge within 100000 samples (final confidence ~ 0.864), while for IS increasing the accuracy does not require higher overhead, allowing it to converge faster than Bayesian. For this problem, a higher N_I would be a reasonable choice, especially for low accuracy.
- Windy is similar to Sorting, since there are many paths, all with comparable probability, and only a few of them are classified as success. However, while for Sorting the Bayesian estimator quickly converged for accuracy 10^{-3} without observing any failure, in this case the probability of success is high enough to allow sampling both types of path. This increases the variance of the sample, slowing down the statistical estimator. On the other hand, for IS, thanks to pruning, as soon as the few success paths are collected they are pruned out, reducing the variance of the samples of subsequent iterations and speeding up convergence.

In summary, IS is particularly effective for problems where a subset of the execution paths accounts for a large portion of the inputs. In this case, such paths are likely to be pruned out after a few iterations increasing the confidence on the partial result. Also, IS outperforms statistical methods when high accuracy is required. Finally, if an exact analysis is required for a problem that would require too much memory to be analyzed with previous approaches [9], IS can analyze them incrementally, producing intermediate results with quantified confidence after each iteration, though usually taking longer time.

Hypothesis testing. The results for hypothesis testing are shown in Table 2. θ and T represent the hypothesis ($H_0 : Pr(P \models \phi) \geq \theta$) and the confidence threshold to accept or reject H_0, and *Result* is the result computed (whether or not the hypothesis holds), while the meanings of the other columns are the same as before. Once again, we assume a uniform usage profile.

Table 2: Hypothesis testing results (* denotes convergence for sufficient exact conditions, ** denotes a false positive/negative)

	θ	T	N_I	Iter	Result	Time
OAE	0.999	10^5	100	2	true	40,150
	0.999	10^5	1000	1	true	35,458
	0.9999	10^5	100	2	true	40,495
	0.9999	10^5	1000	2	true	168,000
	0.999	10^5	100000	–	true	36,295
	0.9999	10^5	100000	–	true	362,125
	θ	T	N_I	Iter	Result	Time
MER (small)	0.74999	10^5	100	4	true*	143,775
	0.74999	10^5	1000	2	true*	541,618
	0.9	10^5	100	1	false*	34,822
	0.9	10^5	1000	1	false	80,266
	0.74999	10^5	100000	–	–	25,763,139
	0.9	10^5	100000	–	false	79,229
	θ	T	N_I	Iter	Result	Time
Sorting	0.999978	10^5	100	47	true	1,567,080
	0.999978	10^5	1000	6	true	1,291,770
	0.999999999	10^5	100	61	false	1,931,810
	0.999999999	10^5	1000	7	false	1,567,732
	0.9999978	10^5	100000	–	–	9,372,129
	0.999999999	10^5	100000	–	false	1,536,449
	θ	T	N_I	Iter	Result	Time
Windy (small)	0.003073625	10^5	100	1	false**	11,120
	0.003073625	10^5	1000	1	true	110,437
	0.004083625	10^5	100	1	false	10,961
	0.004083625	10^5	1000	3	false*	210,286
	0.003073625	10^5	100000	–	true	627,864
	0.004083625	10^5	100000	–	–	6,257,120

Our choices of θ are values close to the actual success probabilities, obtained by estimation and as given in Table 1. As expected [35], hypothesis testing is usually faster than estimation. However, when θ is very close to the actual probability of success, Bayesian methods fail to converge within a reasonable amount of time (results marked with $-$). IS responds to this situation by requiring more iterations (more rounds of sampling/pruning). Compare, for example, the cases with $\theta = .9$ and $\theta = .74999$ for MER (small). IS generally performs better than a pure Bayesian testing (and for some smaller cases the sampling procedure covered, by chance, the full domain after a just few iterations, producing an ex-

act result). Interestingly, in the first experiment reported for Windy (small) with $N_I = 100$ we obtained a false negative result. In this case the Bayesian component of IS converged to a false decision after the 100 samples produced, by chance, 100 failures. Increasing the number of samples N_I was enough to avoid this error.

Table 3: Hypothesis testing results where "classical" symbolic execution runs out of memory (* denotes convergence for sufficient exact conditions)

	θ	T	N_I	Iter	Result	Time
MER (large)	0.2	10^5	100	1	true	55,004
	0.2	10^5	1000	1	true	287,829
	0.35	10^5	100	15	false*	913,109
	0.35	10^5	1000	1	true	287,372
	θ	T	N_I	Iter	Result	Time
Windy (large)	10^{-1}	10^5	100	1	false	30,000
	10^{-1}	10^5	1000	1	false	61,968
	10^{-3}	10^5	100	174	true	6,836,523
	10^{-3}	10^5	1000	7	true	804,979
	10^{-5}	10^5	100	5	true	146,986
	10^{-5}	10^5	1000	1	true	82,998

Intractable "classic" symbolic execution. Table 3 shows the results for a second set of experiments where we ran the techniques on the larger examples for which "classical" symbolic execution is intractable. We show results for the most efficient technique from the smaller cases, i.e., IS for hypothesis testing. There, IS was able to converge to a decision within a reasonable amount of time. Nevertheless, the large number of execution paths of these cases led for MER (Large) with θ close to the actual success probability to a false positive result for $\theta = .35$ and $N_I = 1000$; we know it is a false positive because with $N_I = 100$ we obtained termination for a sufficient condition check. As already discussed, a false positive result is possible for statistical testing. IS can mitigate this issue by leveraging its exact analysis component, as for the case of $N_I = 100$, although, in some cases, even 100 could be enough to make the Bayesian component of IS converge to the wrong conclusion, and an even smaller value for N_I might be required.

Usage profiles. We briefly mention the impact of the usage profiles on the probability of satisfying a target property. We analyzed OAE with a different usage profile, where one input variable (*thrust*) has a Gaussian (normal) distribution. The Gaussian distribution was approximated by discretizing the domain of *thrust* of into 5 segments, which led to 5 usage scenarios with different probabilities [9].

Under this usage profile, the density of inputs following the "normal behavior" paths is reduced, requiring more rounds of pruning for IS estimation to converge, even accuracy as low as 10^{-1}. This results in longer computation time, though still within reasonable ranges. For example, IS with Bayesian estimation for confidence 0.975 took approx. 50,000 ms in 5 iterations (with 100 or 1000 samples per iteration) while for confidence 0.99 it took approx. 167,000 ms in 6 iterations.

The source code for all the examples (except OAE) and more experimental data are available from [8].

7. RELATED WORK

Our work is related to statistical model checking (SMC) [32], also formulated as a statistical hypothesis testing problem verified through Wald's sequential probability ratio test (SPRT) [29]. SPRT does not fix the required number of samples a priori but uses a sequential approach to decide after each sample whether to stop or continue. A different hypothesis testing criterion has been proposed in [26], where the size of the sample set is automatically increased until it allows for satisfying the convergence criteria. In [13], SMC has been formulated as an estimation problem, with the number of samples fixed a priori by means of the Chernoff and Hoeffding bound [14]. Other approaches for deciding the number of samples have been discussed in [26, 34]. Some of these approaches have been implemented in well-known probabilistic model checkers [16, 30].

In our work we combined Bayesian inference techniques with exact analysis through the IS technique, which is shown to provide better performance than the pure Bayesian analysis.

A recent approach related to ours [19] provides automated reliability estimation over partial systematic explorations applied to models. The approach first performs sampling over the model and then applies invariant inference over the samples. The inferred invariant characterizes a partial model which is then exhaustively explored using (exact) probabilistic model checking, obtaining better results than (full model) probabilistic and statistical model checking for system models.

The techniques we propose are different. Indeed we focus on the use of symbolic execution to analyze software from its source code, while [19] focuses on Markov chain models analyzed through probabilistic model checking. The samples in [19] are used to produce an approximate simplified model to be analyzed, while instead we use an iterative process that prunes the execution tree and guides the sampling towards low-probability paths.

We proposed several techniques for the probabilistic analysis of programs [2, 9, 10]. The approaches in [9, 10] can only perform exact analysis that requires all paths to be evaluated. The work in [2] addresses the approximate analysis of non-linear constraints; we can apply the techniques described here also in that domain, using the quantification procedure from [2] instead of model counting. Another approximate analysis for programs is proposed in [25]; that also uses sampling of symbolic paths (but no incremental or informed sampling as we do here) and gives bounds on the probability of events of interest in a program. In more recent work we study statistical techniques that target specifically programs that have nondeterminism (for example due to concurrency) [18]. The work also uses hypothesis testing (a simpler form than here) but its main focus is on deriving optimal schedulers, with the best technique using reinforcement learning for the most promising scheduler moves.

Our work shares similar goals with guided testing techniques, which provide heuristics to guide the exploration of a program towards "interesting" paths (to increase coverage or to uncover errors), e.g., [4, 27] and many other works. However such techniques do not provide statistical guarantees as we do here.

8. CONCLUSIONS

We described statistical symbolic execution, for the analysis of software implementations. The technique uses a randomized sampling of symbolic paths with Bayesian estimation and hypothesis testing. We also proposed Informed Sampling, an iterative approach that first explores the paths with high statistical significance, prunes them from the state space and then keeps guiding the execution along less likely paths. Informed sampling combines statistical information from sampling with exact analysis for pruned paths leading to provably improved convergence of the statistical analysis. The techniques have been implemented in the context of Symbolic PathFinder and have been shown to be effective for the analysis of Java programs. In the future we plan to perform further evaluations and to investigate applications in statistical information flow analysis. We also plan an in-depth study on probability computations for programs with structured inputs.

9. REFERENCES

[1] D. Balasubramanian, C. S. Psreanu, G. Karsai, and M. R. Lowry. Polyglot: Systematic analysis for multiple Statechart formalisms. In *TACAS*, LNCS #7795, pages 523–529, Mar. 2013.

[2] M. Borges, A. Filieri, M. d'Amorim, C. S. Păsăreanu, and W. Visser. Compositional solution space quantification for probabilistic software analysis. In *PLDI*, 2014 – to appear.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. *SIGSOFT Software Engineering Notes*, 27(4):123–133, July 2002.

[4] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE*, pages 443–446, Sept. 2008.

[5] R. Chambers and R. Clark. *An Introduction to Model-Based Survey Sampling with Applications*. Oxford Statistical Science Series. OUP Oxford, 2012.

[6] J. A. De Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. Effective lattice point counting in rational convex polytopes. *Journal of Symbolic Computation*, 38(4):1273–1302, Oct. 2004.

[7] A. Filieri, C. Ghezzi, and G. Tamburrelli. A formal approach to adaptive software: continuous assurance of non-functional requirements. *Formal Aspects of Computing*, 24(2):163–186, Mar. 2012.

[8] A. Filieri, C. S. Păsăreanu, and W. Visser. Statistical analyzer for SPF. `http://www.iste.uni-stuttgart.de/rss/people/filieri/2014-fse-jpf`.

[9] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in Symbolic PathFinder. In *ICSE*, pages 622–631, July 2013.

[10] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *ISSTA*, pages 166–176, July 2012.

[11] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian data analysis*. CRC press, 2003.

[12] J. H. Halton. A retrospective and prospective survey of the Monte Carlo method. *Siam review*, 12(1):1–63, Jan. 1970.

[13] T. Herault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In *VMCAI*, LNCS #2937, pages 73–84, Jan. 2004.

[14] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, Mar. 1963.

[15] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[16] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, LNCS #6806, pages 585–591, July 2011.

[17] D. V. Lindley. The present position in Bayesian statistics. *Statistical Science*, 5(1):44–89, Mar. 1990.

[18] K. Luckow, C. S. Păsăreanu, M. Dwyer, A. Filieri, and W. Visser. Probabilistic symbolic execution for nondeterministic programs. In *CAV*, 2014 – submitted.

[19] E. Pavese, V. A. Braberman, and S. Uchitel. Automated reliability estimation over partial systematic explorations. In *ICSE*, pages 602–611, July 2013.

[20] W. R. Pestman. *Mathematical Statistics*. De Gruyter Textbook. De Gruyter, 2009.

[21] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, pages 15–26, July 2008.

[22] C. S. Păsăreanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, Sept. 2013.

[23] C. Robert. *The Bayesian choice: from decision-theoretic foundations to computational implementation*. Springer-Verlag, 2007.

[24] C. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer Texts in Statistics. Springer-Verlag, 2010.

[25] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *PLDI*, pages 447–458, June 2013.

[26] K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In *CAV*, LNCS #3576, pages 266–280, July 2005.

[27] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. eXpress: guided path exploration for efficient regression test generation. In *ISSTA*, pages 1–11, July 2011.

[28] UC Davis, Mathematics. LattE. | http://www.math.ucdavis.edu/ latte.

[29] A. Wald. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186, June 1945.

[30] H. L. S. Younes. Ymer: A statistical model checker. In *CAV*, LNCS #3576, pages 429–433, July 2005.

[31] H. L. S. Younes, M. Kwiatkowska, G. Norman, and D. Parker. Numerical vs. statistical probabilistic model checking. *International Journal on Software Tools for Technology Transfer*, 8(3):216–228, June 2006.

[32] H. L. S. Younes and D. J. Musliner. Probabilistic plan verification through acceptance sampling. In *AIPS-02 Workshop on Planning via Model Checking*, pages 81–88, Apr. 2002.

[33] H. L. S. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *CAV*, LNCS #2404, pages 223–235, July 2002.

[34] P. Zuliani, C. Baier, and E. M. Clarke. Rare-event verification for stochastic hybrid systems. In *HSCC*, pages 217–226, Apr. 2012.

[35] P. Zuliani, A. Platzer, and E. M. Clarke. Bayesian statistical model checking with application to Stateflow/Simulink verification. *Formal Methods in System Design*, 43(2):338–367, Oct. 2013.