Precise Data-Driven Approximation for Program Analysis via Fuzzing

Nikhil Parasaram[†]
ucabnp1@ucl.ac.uk
UCL, UK

Earl T. Barr
e.barr@ucl.ac.uk
UCL, UK

Sergey Mechtaev
s.mechtaev@ucl.ac.uk
UCL, UK

Marcel Böhme
marcel.boehme@acm.org
MPI-SP, Germany

***Abstract*—Program analysis techniques such as abstract interpretation and symbolic execution suffer from imprecision due to over- and underapproximation, which results in false alarms and missed violations. To alleviate this imprecision, we propose a novel data structure, program state probability (PSP), that leverages execution samples to probabilistically approximate reachable program states. The core intuition of this approximation is that the probability of reaching a given state varies greatly, and thus we can considerably increase analysis precision at the cost of a small probability of unsoundness or incompleteness, which is acceptable when analysis targets bug-finding. Specifically, PSP enhances existing analyses by disregarding low-probability states deemed feasible by overapproximation and recognising high-probability states deemed infeasible by underapproximation. We apply PSP in three domains. First, we show that PSP enhances the precision of the Clam abstract interpreter in terms of MCC from 0.09 to 0.27 and F1 score from 0.22 to 0.34. Second, we demonstrate that a symbolic execution search strategy based on PSP that prioritises program states with a higher probability increases the number of found bugs and reduces the number of solver calls compared to state-of-the-art techniques. Third, a program repair patch prioritisation strategy based on PSP reduces the average patch rank by 26%.**

I. Introduction

Program analysis checks if a given program satisfies certain correctness properties. Due to the undecidability, program analysers approximate program behaviour, which results in imprecision. Techniques like abstract interpretation [1] overapproximate program behaviour, causing false positives (false alarms), and techniques like symbolic execution [2], [3] underapproximate program behaviour, which leads to false negatives (missed violations). Abstract interpretation resorts to coarse abstractions of program states to make their analysis decidable, at the cost of precision. On the other hand, symbolic execution is imprecise because it explores a finite number of execution paths and thus may miss important behaviours.

To improve the precision and recall of program analysis, we propose an alternative to underapproximate and overapproximate reasoning. Our approach leverages execution samples to probabilistically approximate program states via a novel data structure called *program state probability* (PSP). For each state, PSP estimates the probability of the existence of a program input that reaches that particular state during execution. We realise PSP by training a model, such as random forest, on data obtained from fuzzing.

Our technique's fundamental premise is that significant variation exists in the likelihood of reaching various states. Therefore, we can considerably enhance analysis precision at the cost of a small probability of unsoundness or incompleteness. First, PSP enables us to disregard low-probability states deemed feasible by overapproximation, thus reducing false positives. Second, PSP enables us to prioritise high-probability states falling outside of an underapproximation, thus reducing false negatives. The level at which low-probability states are ignored and high-probability, yet unanalysed by an underapproximation, states are prioritised is controlled by a user-defined threshold, which functions as a control knob for this trade-off. We demonstrate PSP's utility by applying it to tackle challenges faced by two widely-used analysis techniques — abstract interpretation and symbolic execution. PSP reduces abstract interpretation's false positives and increases the number of violations detected by symbolic execution. Additionally, we apply PSP to test-driven program repair to prioritise correct patches, thereby alleviating test-overfitting.

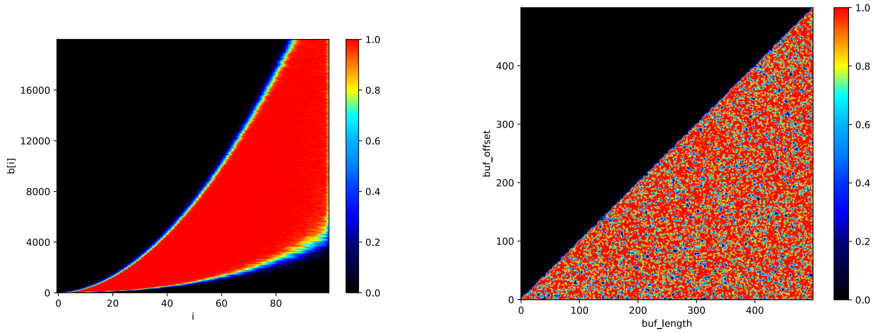
Abstract interpretation [1] approximates program states using abstract domains to make analysis scalable, which often leads to false positives, causing developers to ignore legitimate warnings [4]. Minimising false positives is crucial to ensure the usefulness of abstract interpretation. To achieve that, we propose augmenting abstract domains with program state probabilities constructed using values from fuzzing. PSP allows us to disregard low-probability states deemed reachable by the abstraction. Although this creates a low probability of unsoundness, we demonstrate that PSP effectively reduces false positives, which is important for the practical use of abstract interpretation as a bug-catching tool.

Symbolic execution executes a program with symbolic inputs and creates path conditions for the explored paths, which are solved using an SMT solver to identify feasible paths and potential bugs [5]. However, it suffers from a scalability bottleneck due to the high computational cost of constraint solving. We applied PSP to reduce the solving cost by identifying probable states and guessing which path conditions are satisfiable. Specifically, we prioritise exploring paths that are more likely to be feasible and skip solving path conditions whose probability of being satisfiable is above a user-defined threshold. As a result, PSP facilitates deeper code exploration, enabling symbolic execution to find bugs faster.

Program repair tools face the challenge of searching for a

[†] This work was initiated during the author's internship at MPI-SP.

```
int N, off;
scanf("%d", &N);
if (N <= 0) return 1;
ll *a = malloc(sizeof(ll) * N);
ll *b = malloc(sizeof(ll) * N);
for (int i = 0; i < N; i++) {
    scanf("%d", &off);
    if (off < 0 || off > 6)
        return 1;
    int p = i - 1;
    a[i] = (i == 0) ? 6 : a[p] + off;
    b[i] = (i == 0) ? 0 : b[p] + a[p];
    assert(b[i] <= 3*i*i + 3*i);
}
return 1;
```



(a) A program from SV-benchmarks for which Clam produces a false positive (`ll` denotes **`long long`**).

(b) A heatmap showing the probability of reaching programs states *w.r.t.* the values of `b[i]` and `i` (cooler means less likely).

(c) A heatmap computed when analysing an assertion in OpenSSL, for which Clam generates a false positive.

Fig. 1: An example program, a PSP heatmap computed for this program, and a heatmap for analysis of OpenSSL.

correct patch in a search space that contains many irrelevant or incorrect patches. One way to address this challenge is to prioritise patches in a way that increases the likelihood of finding a correct patch. We formulate a new hypothesis that the set of values most variables in a program can take is invariant with respect to small edits to the program. Relying on this hypothesis, we prioritise patches that make minimal changes to the range of all values program variables can take during execution. PSP enables us to efficiently implement this prioritisation strategy by probabilistically quantifying unchanged bindings of variables to values.

We implemented PSP using AFL fuzzer [6] for C programs, and using Fuzzer Harvey [7] for smart contracts. For abstract interpretation, we integrated PSP with Clam analyser [8] for C programs. An evaluation on seven programs from Magma benchmark [9] revealed that PSP enhances the precision of Clam in terms of MCC from 0.09 to 0.27 and F1 score from 0.22 to 0.34. For symbolic execution, we compared our PSP-based search strategy with the state-of-the-art search strategy using pending constraints [10] and abstract symbolic execution (ASE) [11]. To compare with the pending constraints, we implemented our strategy in Mythril [12] symbolic executor for smart contracts, and to compare with ASE, we implemented PSP inside the ASE tool. Our experiments demonstrated that PSP increases the number of bugs found by 4.1% compared to the pending constraints, and reduces the number of solver calls by 77 times compared to ASE. For program repair, we integrated PSP with the state-of-the-art patch prioritisation strategy of Rete [13]. For bugs from ManyBugs benchmark [14], PSP decreased the average rank of correct patches by 26%.

In this paper, we make the following contributions:

- Introduce program state probability (PSP), which probabilistically approximates program states.
- Apply PSP to reduce false positives during static analysis.
- Propose a search strategy relying on PSP to guide symbolic execution to find more bugs.
- Propose patch prioritisation strategy using PSP to effectively prioritise correct patches for program repair.

PSP's implementation, and the scripts and data used to evaluate it can be found in the accompanying package https://zenodo.org/record/7902213.

II. Overview

In this section, we discuss the general intuition behind program state probability, and illustrate one of its applications: reducing false positives of static analysis.

Figure 1a depicts a simplified code fragment from SV-benchmarks, which serves to evaluate the accuracy of program analysis tools. The program comprises a sequence of memory allocations and assignments, and includes a loop and an assertion. The assertion checks whether `b[i] <= 3*i*i + 3*i` holds for all possible inputs at the end of the program. Clam [8], an LLVM-based abstract interpreter for C programs, imprecisely analyses the code, yielding intervals like $[8, \infty]$ for `b`, which is inaccurate and leads to a false positive. Notably, the interval domain does not capture `b[i]`'s dependency on `i`.

To overcome this issue, we augment Clam's approximation with a probability distribution constructed from fuzzing data. Figure 1b visusalises PSP trained on fuzzing data as a heatmap over program states, wrt the bindings of the variables `b[i]` and `i`. Cooler colours represent a lower probability, under the fuzzing campaign, that the program can generate such a program state. The heatmap reveals a more precise range of values for b than Clam, while still capturing the dependencies between `b[i]` and `i`.

The upper curve in the heatmap corresponds to the function `3*i*i + 3*i`, which satisfies the assertion. The lower bound provided by the heatmap is close to the actual lower bound, which is `6i`. At large values of `i`, the lower bound does not precisely match the lower bound generated by PSP because we did not provide enough seed inputs for the fuzzer. However, it is sufficient to conclude that the probability that there exists an input that leads to an assertion violation is 0.1826, which is below our default threshold of 0.8. Hence, PSP helps to eliminate Clam's false positive.

The heatmap in Figure 1c is obtained by analysing a larger program, OpenSSL, which contains `assert(ctx->buf_len >= ctx->buf_offset)`. The heatmap shows the distribution of values for two variables in this assertion. We can see that the heatmap's heat extends up to the assertion boundary (*i.e.* `ctx->buf_len = ctx->buf_offset`), indicating that the assertion holds with a high probability under PSP. In contrast, Clam does not capture the relationship between these variables, which leads to a false positive.

III. Program State Probability

This work rests on the idea of using the probability that a program can reach a given program state to precisely approximate program behaviour. Here, we formalise program state probability, detail how to calculate it, and apply it to estimate the likelihood of a program condition being satisfiable.

We use the following terminology. Let I be the set of all possible inputs a program f can take. Let V be f's variables and X be the disjoint union over the values that variables can take. Let Σ^f be the set of all concrete states the program f can potentially take. Each state $\sigma : V \rightarrow X \in \Sigma$ maps variables to values. We assume that V contains the program counter, which binds each state to the location in the program at which it is computed. Consider the set of abstract states (sets of concrete states) computed during an abstract interpretation of the program f. Let A_f be the union of all the abstract states computed during the abstract interpretation of f, so $A_f \subseteq \Sigma^f$.

A. Estimating Program State Probability

Let f be a program, let $\sigma \in \Sigma^f$ be an arbitrary program state of f, and let $f(i) = \sigma_0, \sigma_1, \ldots, \sigma_n = \overline{\sigma}$ denote the execution of f instrumented to output its entire state trajectory. This indicator function defines the set of states f can generate:

$$\mathbb{I}_{\Sigma^f}(\sigma) = \begin{cases} 1 & \text{if } \exists i \in I, \sigma \in \{\sigma_j \mid f(i) = \overline{\sigma} \wedge j \in [0..|\overline{\sigma}|]\} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

This indicator function is not computable, so we define program state probability to approximate it.

Definition 1 (Program State Probability (PSP)): For the program f, the *program state probability* of the program state $\sigma \in \Sigma^f$ is the probability that f can generate σ. Formally, it is $P(\mathbb{I}_{\Sigma^f}(\sigma) = 1)$.

To estimate PSP, we use the information generated from fuzzing. We represent the event of running a fuzzer as F and the fuzzing data using the multiset *Fuzz* whose element multiplicity represents the number of times the program state σ was encountered n times during the fuzzing campaign.

We use the *Fuzz* produced by fuzzing campaign F to estimate PSP as follows:

$$P(\mathbb{I}_{\Sigma^f}(\sigma) \mid F) = \begin{cases} 1 & \sigma \in \mathit{Fuzz} \\ 0 & \sigma \notin A_f \\ r(\sigma, \mathit{Fuzz}) & \text{otherwise} \end{cases} \tag{2}$$

where r is a probability distribution, which defaults to uniform, over the states unseen during the fuzzing campaign F.

When the fuzzing campaign encountered σ, we can say with certainty that $P(\mathbb{I}_{\Sigma^f}(\sigma) \mid F) = 1$. When $\sigma \notin A_f$, we can say with certainty that $P(\mathbb{I}_{\Sigma^f}(\sigma) \mid F) = 0$, as abstract domain overapproximates the set of possible program states. Otherwise, we use the function r to estimate, or “radiate”, probability f can generate σ from the observations in *Fuzz*. This function can be instantiated in various ways, as discussed in Section III-B. Where clear from context, we use $P(\mathbb{I}_{\Sigma^f}(\sigma))$ to denote $P(\mathbb{I}_{\Sigma^f}(\sigma) \mid F)$.

B. Estimating Unseen States

Empirically, we observe that variable bindings often obey rules and exhibit patterns, like monotonically increasing at a fixed stride, only taking on odd (or even) numbers, or oscillating among a few error codes. We leverage this insight to estimate $r(\sigma, \mathit{Fuzz})$. A program generates very few of its possible program states. We first check whether σ's neighbourhood has enough data to contend with this sparsity. If it does, we look for patterns in it. If we discern a pattern, we increase the probability of unseen values that obey the pattern. We use smoothing to reserve probability weight for the rest of the values.

Let σ's k-neighbors in *Fuzz* be

$$N(\sigma, k) = \{\sigma' \in \mathit{Fuzz} \mid d(\sigma, \sigma') \leq k\}$$

where d is a distance function over program states. When the population of a state's neighbourhood exceeds the sample threshold u, we train a supervised model M_s to estimate $r(\sigma, \mathit{Fuzz})$ and use it predict states in a given observed state's neighbourhood that were unseen during fuzzing. Otherwise, we resort to the unsupervised statistical method $M_{\overline{s}}$:

$$r(\sigma, \mathit{Fuzz}) = \begin{cases} M_s\big(N(\sigma, k_s), \sigma\big) & |N(\sigma, k_s)| \geq u \\ M_{\overline{s}}\big(N(\sigma, k_{\overline{s}}), \sigma\big) & \text{otherwise} \end{cases} \tag{3}$$

where the distances k_s and $k_{\overline{s}}$ are distinct because they may need to differ.

Various models can be used to realise M_s. In Section V, we describe how we built a random forest to detect the simple patterns that we observed in our data.

To estimate $M_{\overline{s}}\big(N(\sigma, k_{\overline{s}}), \sigma\big)$, we can subject the empirical distribution that the multiset *Fuzz* defines to various classical smoothing techniques such as kernel density estimation. Alternately, we could use heat diffusion to distribute probabilities across neighbouring points:

$$\frac{\partial P(\mathbb{I}(\sigma))}{\partial t} = \nabla^2 P(\mathbb{I}(\sigma)) + Q(\sigma, t) \tag{4}$$

The variable t represents the time during which the system evolves. The function $Q(\sigma, t)$ supplies or removes heat from the system, ensuring that the heat sources and sinks (*i.e.*, states encountered during fuzzing and states not in the abstract domain) maintain their probability. Specifically, $\forall t \in [0, \infty]$, $\sigma \in \mathit{Fuzz} \Rightarrow P(\mathbb{I}(\sigma \mid F)) = 1$ and $\sigma \notin A \Rightarrow P(\mathbb{I}(\sigma \mid F)) = 0$. We add additional constraints which preserve the probability axioms (*i.e.* $0 \leq P(\mathbb{I}(\sigma)) \leq 1$). We numerically solve Equation (4) using the Forward Time Centered Space method [15]. We evaluated these different approaches in Section VI-E; our implementation uses the heat diffusion method because it worked best Section V.

C. Satisfiability Under PSP

Consider an arbitrary condition C in the program f containing variables $v_1 \dots v_n$. We denote the probability that there exists an input that satisfies this condition as $P(\mathbb{I}_C^f)$ where:

$$\mathbb{I}_C^f = \begin{cases} 1 & \text{if } \exists \sigma \in \Sigma^f, \mathbb{I}_{\Sigma^f}(\sigma) = 1 \wedge \langle C, \sigma \rangle \Downarrow 1 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where $\Downarrow$ evaluates C in the program state σ.

We compute the lower and upper bounds using the following equations:

$$P(\mathbb{I}_C^f) \geq \max_{\sigma \in \Sigma_C^f} P(\mathbb{I}_{\Sigma^f}(\sigma)) \tag{6}$$

$$P(\mathbb{I}_C^f) \leq \min\big(1, \sum_{\sigma \in \Sigma_C^f} P(\mathbb{I}_{\Sigma^f}(\sigma)\big) \tag{7}$$

To weakly approximate $P(\mathbb{I}_C^f)$, we assume that the events in $\Sigma_C^f = \{\mathbb{I}_\Sigma^f(\sigma_i) \mid \sigma_i \in \Sigma, \langle C, \sigma_i \rangle \Downarrow 1\}$ are mutually independent:

$$P(\mathbb{I}_C^f) \approx 1 - \prod_{\sigma_i \in \Sigma_C^f} \Big(1 - P\big(\mathbb{I}_{\Sigma^f}(\sigma_i)\big)\Big) \tag{8}$$

The satisfiability conditions can be computationally expensive when computing $P\big(\mathbb{I}_{\Sigma^f}(\sigma)\big)$ for each state $\sigma \in \Sigma_C^f$ using Equation (3). However, under certain practical constraints, they become inexpensive to compute. We show how we efficiently and effectively approximate it in Section V.

IV. PSP Applications

Here, we introduce three applications of PSP: enhancing the precision of static analysis, optimising symbolic execution, and improving the quality of automatically generated patches.

A. Abstract Interpretation

Abstract interpretation overapproximates program states to make analysis tractable and scalable, which leads to false positives, discouraging developers from examining analysis violations. Therefore, minimising false positives is crucial for the usability of abstract interpretation. To achieve that, we use PSP to construct $P(\mathbb{I}(\sigma))$ by relying on fuzzing data as well as the abstract states A_f computed by an abstract interpretation tool. Then, we calculate $P(\mathbb{I}_{\neg C}^f)$ for each assertion C, as explained in Section III-C. We only report an assertion violation if the resulting probability is greater than a user-defined threshold θ. This enables us to disregard low-probability states deemed reachable by overapproximation, thus reducing false positives.

We define the threshold θ such that $\theta = 0 \implies PSP$ is a conservative overapproximation and $\theta = 1 \implies PSP$ is a conservative underapproximation. Raw PSP can violate θ's definition, since it can predict $P(\mathbb{I}_{\neg C}^f) = 1$ for an assertion C which is always satisfied and $P(\mathbb{I}_{\neg C}^f) = 0$ for an infeasible assertion. To prevent these cases, we restrict PSP to estimate probabilities in the interval $(0, 1)$, not $[0, 1]$. To achieve this, we post process PSP's estimates: we rewrite its 0 estimates to $0 + \epsilon$ and its 1 estimates to $1 - \epsilon$. This postprocessing enforces soundness at $\theta = 1$, ensuring we do not report false positives due to imprecise estimates.

B. Symbolic Execution

Symbolic execution constructs a path condition, the conjunction of conditionals it encounters along a path. It queries an SMT solver to determine which paths are feasible by checking the satisfiability of their path conditions, which is computationally expensive. PSP can reduce this cost by identifying probable states and guessing if path conditions are satisfiable.

We utilise PSP to prioritise execution paths when symbolically executing a subject program to find bugs. The algorithm is shown in Algorithm 1. As usual, it takes the subject program and a termination condition, which could bound steps, time, memory, or computation. Algorithm 1's key novelties are two-fold: 1) it prioritises probable paths, line 10, and 2) it skips, line 18, solving probable program states, only checking whether improbable states are satisfiable (line 19). The $solve$ helper function extracts the path condition from the symbolic state s and invokes an SMT solver. The $bugs(s)$ function checks whether the symbolic state s can possibly break certain criteria drawn from the semi-universal implicit specification of most programs, such as integer or buffer overflows.

Because of these two features, Algorithm 1 spends more time exploring probable paths than solving constraints. Despite not solving probable states, Algorithm 1 always explicitly solves and stores the inputs for those bugs it finds, *i.e.*, $solve(s) \neq \text{SAT} \implies bugs(s) = \emptyset$. It may, of course, deem some UNSAT state to be probable, thus treating it as SAT and end up making redundant calls. To support its new features, Algorithm 1 additionally takes, as input, a PSP model trained on a fuzzing campaign F's fuzzing data and a threshold value θ that determines which program states are sufficiently improbable to ignore. Finally, it returns, as usual, the set of bugs it finds during its run.

We cannot directly employ the bounds discussed in Section III-C as it is not feasible to enumerate all possible program states. Symbolic execution constructs a path condition PC in conjunctive normal form that conjuncts all the conditions it encounters along a path, which we leverage to reduce the space of program states. Upon encountering the condition C, a symbolic executor extends its current path condition via $PC' = PC \wedge C$. To compute PC's probability, we need to compute the probability of $P(PC \wedge C) = P(C \mid PC)P(PC)$. We already know $P(PC)$, since we computed it the last time we extended it. Some of the clauses, or conjuncts, in PC are independent of the clauses in C. Since $PC = c_1 \wedge c_2 \wedge \ldots \wedge c_n$, we can rewrite $P(C \mid PC)$ as $P(C \mid c_1, c_2, \ldots, c_n)$. By the definition of independence, $P(X \mid Y) = P(X)$. We leverage this

Algorithm 1: Symbolic execution using PSP.

```
**Input:**
$f$: The subject program
$PSP$: The PSP model built from fuzzing data
$\theta$: Threshold for ignoring improbable states
$\kappa$: Termination criteria
**Output:**
$B$: The set of bugs discovered

$S_0 := \mathit{initState}(f)$
$\mathit{workList} :=$ PriorityQueue($PSP$)
$\mathit{workList}$.put($S_0$)
$B := \{\}$
while $\mathit{workList} \neq \emptyset \wedge \kappa$ do
    $s := \mathit{workList}$.next()
    $B := B \cup \mathit{bugs}(s)$
    $\mathit{newStates} := \mathit{execute}(s)$
    for $s \in \mathit{newStates}$ do
        if $PSP(\mathbb{I}^f_{s.pc}) < \theta$ then
            if $\mathit{solve}(s) \neq \mathit{SAT}$ then
                continue
        $\mathit{workList}$.put($s$)
return $B$
```

fact to simplify estimating $P(C \mid c_1, c_2, \ldots, c_n)$ by dropping all the c_i clauses that are independent of C.

To this end, we employ the χ^2 test of independence to identify and prune irrelevant clauses. We compute the independence of variables from the fuzzing data offline, then use it to infer the independence of clauses. We then eliminate conjuncts from PC that are transitively independent of the clauses in C. Let C' be the conjunction of remaining clauses. The next step is to filter the *Fuzz* data to include only program states generated by code guarded by C'. This is accomplished by recording the path travelled by the fuzzer to discover a program state. The result is $\mathit{Fuzz}' \subseteq \mathit{Fuzz}$. We then use Fuzz' to evaluate $P(C \mid C')$ using Equation (6), which provides the most conservative bound and prioritizes precision. This choice reduces the chance that PSP gives an unfeasible path a high probability of SAT.

C. Patch Prioritisation

Program repair tools search for a patch in a search space that includes many irrelevant or incorrect patches. Due to the sparsity of useful patches, brute force enumeration is usually infeasible. So, patch prioritisation seeks to order the patches so that the correct ones appear earlier during enumeration.

Since program repair tools typically generate patches, whose size is insignificant *w.r.t.* the size of the program, we formulate the following hypothesis:

Hypothesis 1: The set of values that most variables in a program can take is invariant *w.r.t.* the edits δ iff δ changes a sufficiently small portion of the program.

We acknowledge that counterexamples for this hypothesis exist. Nonetheless, we observe that it often holds in practice. Relying on this hypothesis, we prioritise patches that make minimal changes to the range of values program variables take during execution. We speculate this prioritisation strategy can be combined with other strategies to increase the quality of automatically generated patches.

In practice, a patch δ usually replaces one variable with another with an indistinguishable distribution over its values. Thus, our focus here is on probabilistically quantifying probabilistically unchanged bindings. To probabilistically check Hypothesis 1, we first define the subset of a program's state on which an arbitrary predicate holds:

$$\mathbb{I}_{\Sigma^f}(\alpha) = \begin{cases} 1 & \text{if } \{\sigma \mid \sigma \in \Sigma^f, \alpha(\sigma)\} \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

We are interested in computing the probability that the program patch δ does not change some binding when $\delta(f)$ is executed. For the fixed binding $v = x$, we leverage this indicator function over predicates to define

$$D(v, x) = \left| P(\mathbb{I}_{\Sigma^f}(\sigma(v) = x)) - P(\mathbb{I}_{\Sigma^{\delta(f)}}(\sigma(v) = x)) \right|$$

This difference captures the probability that the two program variants disagree on whether a program state exists with the binding $v = x$; when they are in perfect agreement, this difference is zero. In general, δ can add or remove variables from f, so that V may not equal $V_{\delta(f)}$, the variables in the patched version of f. We define the probability that the patch δ changes variable bindings as:

$$\mathbb{P}(f, \delta) = \prod_{v \in V \cap V_{\delta(f)}} \prod_{x \in X} \big(1 - D(v, x)\big) \tag{10}$$

where v ranges over $V \cap V_{\delta(f)}$ to restrict this computation to shared variables as required by the definition of D.

We cannot directly compute Equation (10), because X is often infinite. Hence, we approximate X with the values observed during the execution of a test suite T on f and $\delta(f)$. We denote these constrained sets as $X|_{T(f)}$ and $X|_{T(\delta(f))}$ and constrain x to their union.

Constraining X alone is not enough to efficiently compute Equation (10). To achieve practical efficiency, we also need to constrain $V \cap V_{\delta(f)}$ in Equation (10)'s outer product. Hence, we constrain v only to variables appearing in the patch δ, which we denote as V_δ. This greatly reduces the number of variables PSP considers. Its use comes at a cost, however: it introduces imprecision because it blinds the computation of Equation (10) to patches that do not directly change an assignment, such as those that modify only a condition that causes an assignment that falls outside of δ to execute and change a binding.

We now show how to leverage these restrictions to speed Equation (10)'s computation. Below, we use b to denote the binding "$\sigma(v) = x$". First, we speed D's computation by precomputing $P(\mathbb{I}_{\Sigma^f}(b))$, using Equation (9). Next, we approximate D's $P(\mathbb{I}_{\Sigma^{\delta(f)}}(b))$ with

$$P(\mathbb{I}_{\Sigma^{\delta(f)}}(b)) \approx \begin{cases} 1 & x \in X|_{T(\delta(p))} \\ P(\mathbb{I}_{\Sigma^f}(b)) & \text{otherwise} \end{cases} \tag{11}$$

because if $x \in X|_{T(\delta(p))}$, we have observed it in a trace and therefore its probability to occur is 1; otherwise, we assume that its probability is same as $P(\mathbb{I}_{\Sigma^f}(b))$, under Hypothesis 1.

We use a case analysis and Equation (11) to transform Equation (10) to

$$\mathbb{P}(f, \delta) = \prod_{v \in V_\delta} \prod_{x \in X|_{T(\delta(f))}} \big(1 - |1 - P(\mathbb{I}_{\Sigma^f}(b)|\big). \tag{12}$$

Equation (12) can be used alongside other probabilistic prioritisation techniques, such as Rete, via the total theorem of probability [16].

V. Implementation

We now detail the interesting design decisions we made to order to realise an analysis framework resting on PSP. First, we discuss the fuzzing tools we used for C and Solidity. We then discuss the exact models and techniques we used to realise Equation (3) and estimate unseen program states. We close by describing how we integrated PSP into the Mythril symbolic execution engine and adapted the Trident patch synthesiser to use PSP to prioritise patches.

Logging Fuzzing Information: To realise PSP, we must effectively estimate $P(\mathbb{I}_{\Sigma^f}(\sigma))$. For C programs, we use AFL [6] to generate values bound to program variables, then use these values to define an empirical probability distribution, which we use as our estimate. To log the bindings that the fuzzer observes along with the line numbers, we instrument each variable with a function that records the variable to a global dictionary flushed to stdout at every return/exit point. For smart contracts, we use the proprietary Fuzzer: Harvey [7] to log variable information. We use the solidity events that a program emits. These events trigger the `LOG` opcode and cause the EVM to add entries to the transaction receipt. Although we do not use `LOG`'s transaction receipt, we do use Harvey to capture logs that satisfy the event signature corresponding to capturing the variable's state and write them to a dictionary that we periodically flush to a file. We can recognise this instrumented event corresponding to capturing the variable's state through its event signature, which is available as input to the `LOG` opcode.

Estimating Unseen States: We use two methods to assign probabilities to unseen program states: a supervised model and an unsupervised statistical method. Since all atomic variable types are bitstrings, which can be interpreted as integers, we estimate unseen states as integers. First, we instantiate the supervised model with Random Forest under the default parameters of scikit-learn. We employ the classic Euclidean distance as the distance function. Our distance function ignores variables that are not shared across the program states. We instantiate u with 100 and k with 20 for computing Equation (3). We use these numbers as they gave good results in small scale experimentation. We train the model on simple patterns (file descriptors, error codes, odd, even, modulo, and other periodic patterns). Since the model is meant to estimate simple patterns, we do not train it to estimate the dependencies across variables and the model considers each variable independently. This makes the model independent of the repository that it is being run on, as the goal is to identify simple patterns, we make use of the same model for all our applications.

To construct the feature vector for a given variable v, we consider the values encountered during fuzzing, in the neighbourhood k as $S = \{n_1, n_2, n_3, \dots n_r\}$. Recall that $\sigma(v) \notin S$, because its current value was not observed during fuzzing. We construct a zero indexed ternary feature vector of size $2k{+}1$ centered around $\sigma(v)$, denoted as $F = [(\sigma(v){-}k) \in S, (\sigma(v) - k + 1) \in S, \dots (\sigma(v) + k) \in S]$. Since $\sigma(v)$ is not in F, we set its value to -1 *i.e.* $F[k] = -1$ to make it easier for the model to differentiate the index of the value it needs to infer. A sample input for requesting to predict the existence of the value 5 for a variable with $S = \{2, 4, 6, 8\}$, $k = 3$ will be $F = \{1, 0, 1, -1, 1, 0, 1\}$.

For the unsupervised statistical method, we employ two methods: kernel density estimation (with scikit-learn's defaults) and the numerical diffusion method. For numerical heat diffusion, we employ a diffusion factor of 6 and numerically compute it for t=10 seconds. We have observed that it works well on ten assertion samples taken from OpenSSL.

Satisfiability Under PSP: The satisfiability conditions can be computationally expensive when computing $P(\mathbb{I}_{\Sigma^f}(\sigma)\big)$ for each state $\sigma \in \Sigma_C^f$ using Equation (3). A query can take one of two paths: $M_s\big(N(\sigma, k_s), \sigma\big)$, when the neighbourhood's density is high, and $M_{\overline{s}}\big(N(\sigma, k_{\overline{s}}), \sigma\big)$ otherwise. A high density neighbourhood, *i.e.* $N(\sigma, k) \geq u$, usually only happens for variables with a small domain (error codes, flags *etc.*) or repeating patterns. As a result, despite being per variable, the computation of $M_s\big(N(\sigma, k_s), \sigma\big)$ is not frequently invoked. Fuzzers explore a finite range of values, typically around $[-10^9, 10^9]$, although, we have usually observed values less than 10^4. We radiate our probability across the states encountered during fuzzing before any queries. Hence, any queries to $M_{\overline{s}}\big(N(\sigma, k_{\overline{s}}), \sigma\big)$ are cheap and instant. The techniques used to radiate the probability, such as kernel smoothing and heat equation, are also inexpensive to compute on this scale.

Symbolic Execution: For symbolic execution, we implement PSP's strategy on Mythril [12], a well-maintained symbolic execution engine for smart contracts. We implement this on Mythril's v0.23.22 version, since earlier versions do not strictly adhere to execution timeout due to Z3 not adhering to its prescribed timeout, which will end up tainting the results. We employ a solver timeout of 25 seconds for all the configurations.

Mythril has two classes of search strategies: ordinary and super. Ordinary strategies have the lowest priority. Super strategies can be stacked on top of them, but not vice versa. The top strategies in the stack, *i.e.* super strategies, have the highest priority. When a strategy makes no choice, the decision is passed down to the next strategy.

We implemented PSP as an ordinary search strategy. The only super strategy on top of PSP is the `BoundedLoopsStrategy`, which, by default, bounds loops to three iterations. We stack `CoverageStrategy` and `BoundedLoopsStrategy` when running the other

TABLE I: The number of bugs/false positives each tool finds. A_{30} denotes running the AFL fuzzer for 30 minutes. $P(F, \theta)$ denotes running the fuzzer F with threshold θ.

Program		Clam		A_{30}		$P(A_{30}, 0.5)$		$P(A_{30}, 0.7)$	
	Bugs	TP	FP	TP	FP	TP	FP	TP	FP
libtiff	14	14	73	1	0	5	8	3	5
libpng	7	7	91	1	0	5	9	5	8
openssl	20	20	173	2	0	8	14	8	11
php	16	16	76	2	0	5	4	5	2
libxml	17	17	83	2	0	7	7	6	4
SQLite	20	20	167	0	0	1	8	0	6
Poppler	22	22	159	1	0	7	16	6	12
Total	116	116	822	9	0	38	67	31	48

TABLE II: Tool performance using IR measures. A_{30} denotes running the AFL fuzzer for 30 minutes. $P(F, \theta)$ denotes running the fuzzer F with threshold θ.

Tools	Precision	Recall	F_1 Score	MCC
Clam	0.12	1.00	0.22	0.09
A_{30}	1.00	0.08	0.14	0.26
$P(A_{30}, 0.1)$	0.19	0.42	0.26	0.13
$P(A_{30}, 0.5)$	0.36	0.33	0.34	0.26
$P(A_{30}, 0.7)$	0.41	0.28	0.34	0.27

baselines on Mythril, such as the pending constraints search strategy [10] and the default *BFS*. We use the `MythrilCoveragePlugin`, a custom Mythril plugin, to record instruction and branch discovery along with their discovery times. For the threshold θ, we employ $\theta = 0.9$, as that provided the best performance out of all thresholds $\theta \in \{\frac{x}{10} \mid x \in \mathbb{Z}, 0 \leq x \leq 10\}$ on a sample of 10 smart contracts taken from SmartBugs-Wild [17] dataset.

A. Patch Prioritisation

To implement patch prioritisation, we extend Trident's [18] patch synthesiser to use PSP to prioritise patches. Trident enumerates every patch to the depth $d = 4$, checking if each patch matches the specification. We modified Trident to use a priority queue instead, ordered by state probability under PSP's $\mathbb{P}$, defined by Equation (12).

VI. Evaluation

We aim to answer the following questions in our evaluation:

RQ1 *How does PSP improve the bug-finding capability of abstract interpretation?*

RQ2 *How does PSP improve the bug-finding capability of symbolic execution?*

RQ3 *How does PSP improve patch prioritisation?*

We also report on PSP's sensitivity to its two critical hyper-parameters, and perform an ablation of its value estimators.

We trained PSP on fuzzing data from coreutils and openssl, split into 90% for training and 10% for testing. We conducted runs on a 16 core, 3.2 GHz machine running Ubuntu 22.10 with 32GB of memory.

To compare PSP with the baselines, we focus on key performance metrics relevant to bug finding tools. First, we present the true positive counts, highlighting the primary objective of bug finding tools — discovering bugs. Second, we report precision, since excessive false positives have a detrimental effect on the usability of bug finding tools [19]. Third, we report the Matthews Correlation Coefficient (MCC), which encapsulates our approach's defect classification capacity, considering the full confusion matrix [20]. We also include the F1 Score to facilitate comparison with related work.

A. RQ1: PSP for Abstract Interpretation

We use the Clam static analyser [8] to overapproximate whether an assertion can be violated and the AFL fuzzer [6] for 30 minutes to under-approximate it. We compare the bugs reported by these tools to the bugs reported by PSP and count the number of false positives or negatives. We sample 7 out of 9 projects from Magma benchmark [9], namely, libtiff, libpng, openssl, php, libxml, Poppler, SQLite. The projects have some explicit assertions that always hold and other assertions that violate to indicate a bug's presence.

Table I shows the number of false positives and false negatives for various tools; Table II shows precision, recall, F1 score, and MCC. As an overapproximate analysis, Clam has the highest recall among all tools. Clam also has the lowest precision: it reports 938 bugs, but only 12% of the reported bugs are actually bugs. In contrast, AFL is under-approximate; it has the highest precision but the lowest recall. After running for 30 minutes, AFL reports only 8% of the 116 bugs.

PSP achieves the highest F1-score, balancing precision and recall. If the threshold value θ is set to 0.5, then PSP finds 38 of the 116 bugs, but it also reports 67 bugs that do not exist. PSP for $\theta = 0.5$ reports substantially more true positives than AFL and substantially fewer false positives than Clam. If we adjust θ to 0.7, the total number of bug reports decreases from 105 to 79, increasing precision from 36 to 41% at the cost of decreasing recall from 33 to 28%.

From Equation (2), we observe that PSP only constructs non-zero probabilities for at least one program state *w.r.t.* a statement iff the statement is covered by the fuzzer. For the statements covered by the fuzzer, PSP at $\theta = 0.7$ has correctly identified, by violating an assertion, 77.5% of the buggy statements as bugs. In contrast, the fuzzer has only identified 18.4% of the buggy statements, which it has covered in the span of 30 minutes as bugs.

RQ1: For the threshold $\theta = 0.7$, PSP finds 31 out of 116 bugs and produces 48 false alarms; it finds 22 more bugs than AFL running for 30m (A_{30}) and does so with 774 fewer false positives than Clam.

In summary, PSP outperforms Clam in terms of precision, and outperforms AFL in terms of the number of found bugs, while outperforming the fuzzer on F1 and effectively matching it on MCC. In practice, analyses like PSP are important for software developers who wish to trade finding more bugs against contending with more false positives.

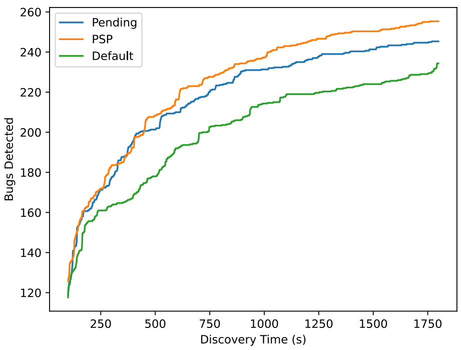


Fig. 2: Bugs Discovered vs Time.

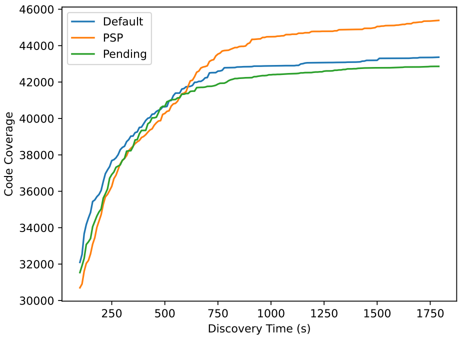


Fig. 3: Instruction Coverage vs Time.

TABLE III: The execution time and the number of solver queries for PSP, ASE and default symbolic execution.

Program	PSP		ASE		baseline	
	time(s)	queries	time(s)	queries	time(s)	queries
bubble_sort_1	8.70	0	8.39	0	38.8	67350
bubble_sort_3	21.70	4644	152.80	637821	266.7	1135634
bubble_sort_all	9.30	0	58.75	40320	59.0	234958
dijkstra	17.40	572	156.60	88726	184.7	99564
dirname	0.06	0	0.05	0	74.7	815764
gcd	1.50	0	125.60	3192	123.6	3192
half	0.01	0	0.01	0	16.4	8010
heap_sort_1	0.78	0	0.81	0	1.6	2340
heap_sort_3	0.74	0	22.50	104524	111.3	518822
heap_sort_all	25.30	9674	288.30	960034	111.1	964444
insert_sort_1	7.50	0	7.40	0	38.6	67350
insert_sort_3	24.90	18268	330.00	1420769	650.8	2860000
insert_sort_all	2.30	3244	15.20	80057	18.5	80638
merge_sort_1	0.20	0	0.20	0	0.4	896.0
merge_sort_3	0.30	0	9.40	29460	34.9	147400
merge_sort_all	0.30	0	18.00	78750	18.9	80638
quick_sort_1	4.70	0	4.80	0	4.9	598.0
quick_sort_3	5.90	0	61.40	205666	128.3	446606
quick_sort_all	5.30	0	25.80	78800	26.7	80638
selection_sort_1	4.10	0	4.10	0	559.7	1192300
selection_sort_3	21.50	70831	771.40	2913602	1048.4	3828538
selection_sort_all	2.30	3244	153.20	521344	153.3	526350
kruskal	12.30	1255	179.90	711343	195.4	789714
bellman-ford	18.50	6632	131.70	91752	150.3	102876
binary_search_all	0.03	0	0.03	0	345.0	8000
linear_find_all	0.01	0	0.01	0	32.5	2000
is_permutation	8.20	6055	383.10	1622733	418.0	1716634
loop_invgen	0.10	0	124.80	22859	246.8	34440
min_max_all	0.50	0	30.20	72067	32.8	77706
fibonacci	0.05	0	0.05	0	6.9	206554
outerproduct	0.08	0	0.08	0	45.0	140616

B. RQ2: PSP for Symbolic Execution

To evaluate PSP's symbolic execution strategy on smart contracts, we compare our strategy (*PSP*) against two baseline strategies: Mythril's default search strategy (*Default*), which is a combination of breadth-first, coverage-directed, and loop bound strategies.[1], and a recent search strategy based on pending constraints [10] (*Pending*), which is implemented on top of Mythril's default search strategy.

We evaluate the symbolic execution strategy of PSP for smart contracts using three sets of 52 random samples extracted from the SmartBugs-Wild [17] dataset, which consists of 47,398 smart contracts from Ethereum mainnet. Smart contracts tend to be smaller than traditional programs due to the limited storage capacity and high cost of running on the blockchain. The bytecode of a smart contract on EVM has a size limit of 24KB [21]. However, smart contracts operate differently from standard programs because they are state machines, where users can modify the state by triggering one of the smart contract's functions. This potentially leads to infinite execution paths if a user repeatedly triggers a function that modifies the state, which results in complex control flow and an increased number of possible execution paths, even for small programs. The average number of lines of a smart contract in our samples is 379.8.

Figure 2 shows the number of bugs found in smart contracts over time by our technique (PSP) and the baseline techniques. Figure 3 shows the number of instructions symbolically executed over time. PSP outperforms the baseline (*Default*) and improved strategy (*Pending*) in terms of number of bugs found. In 30 minutes, PSP can find 9% and 4.1% more bugs than *Default* and *Pending*, respectively. In terms of symbolic coverage of program instructions, PSP performs substantially better than *Default* and *Pending* from 10 minutes onwards. For the first 600 seconds, the coverage-guided *Default* and *Pending* strategies perform better. However, the coverage-guided strategies quickly exhaust the easy-to-discover instructions and start underperforming. PSP's search strategy does not explicitly prioritise coverage. Even though it lags initially due to not focusing on coverage, PSP eventually outperforms the other two strategies due to its superior performance once the easy-to-discover instructions are exhausted.

RQ2 (1/2): PSP's search strategy finds 4.1% more bugs in smart contracts than the state-of-the-art pending constrains search strategy.

Abstract symbolic execution (ASE) [11] represents the amenable portion of the symbolic state using value sets and delegates constraint solving partially to cheaper membership tests in these value sets. For an objective comparison with ASE, we implemented the PSP search strategy in the ASE tool for C programs. Since the programs supported by ASE are smaller, the fuzzer thoroughly covers various paths in that programs, enabling a precise estimation for PSP. Thus, our ASE variant does not implement value sets, as they are superfluous on small programs. We employ two thresholds, 0.1 and 0.9, and only solve path constraints if their estimated probability of satisfiability falls into $(0.1, 0.9)$. We simply

[1] The strategies are called `BFS`, `CoverageStrategy`, and `BoundedLoopsStrategy`.

TABLE IV: Expanded names of the functions used in Table V

Abbreviation	Expansion
RF	Random Forest
DIF	Diffusing the probability
RFDIF	Random Forest on dense data and DIF on sparse data
RFKS	Random Forest on dense data and KDE on sparse data
DIF	Diffusing the probability using Equation (4)
KDE	Kernel Density Estimation

assume the query is UNSAT for a low threshold (< 0.1) and SAT for a high threshold (> 0.9). We construct data by running fuzzing for 3 seconds. For this result, we use the dataset of small programs employed in the original ASE work [11]. The average program size in this dataset is *ca.* 79 lines of code.

Table III shows the results for the comparison of the execution time and the number of solver queries against the ASE tool and the default (baseline) symbolic execution on the benchmark set of C programs provided by the ASE authors. Out of 31 samples tested, PSP required solver intervention in only 10 cases, whereas ASE required intervention in 19 cases. In these 10 cases, we observed that the number of queries sent to the solver was at least an order of magnitude lower than the number of queries sent by ASE. Overall, the total number of queries was reduced by 77 fold when using PSP compared to ASE. These results demonstrate that PSP is a promising approach for optimising symbolic execution, reducing the reliance on solver intervention and improving the efficiency of the technique.

RQ2 (2/2): Compared to ASE, PSP's search strategy reduces the number of solver calls by 77 and the average time of a symbolic execution run by a factor of 15.

C. RQ3: PSP for Patch Prioritisation

We use the C implementation of Rete [13] as the baseline for patch prioritisation. We extend Rete with PSP, using the techniques described in Section IV-C. We chose the C implementation of Rete because the fuzzing framework for C is more robust when compared to Python. We use MB35 dataset [18], consisting of 35 bugs sampled from ManyBugs [14]. Our evaluation of patch prioritisation involves determining the rank of the correct patch in the ordered sequence of generated candidate patches. We restricted our analysis to 8 bugs, excluding those for which Rete does not generate correct patches within 2 hours, as computing precise ranking for such bugs would be impractical. The average patch ranking for Rete and PSP-Rete (Rete augmented with PSP) are 3999 and 2959, respectively, as shown in Table VI.

RQ3: PSP integrated with the state-of-the-art patch prioritisation of Rete helps to decrease the average rank of correct patches by 26% from bugs from ManyBugs.

D. PSP's Hyperparameters

PSP has two critical parameters: The precision threshold θ and the fuzzer configuration A_{time}. Here, we explore PSP's sensitivity to these two parameters on our subset of the Magma benchmark (Section VI-A). The precision threshold of PSP is used for classifying whether a given sample has a bug. Based on how PSP computes probabilities Section III-A, at $\theta = 1$, we have a precision of 1 and at $\theta = 0$, we have a recall of 1. Figure 4 presents the F_1 score and MCC as a function of θ. The F_1 score increases rapidly until $\theta = 0.5$, after which it starts to decrease. This drop in F_1 score from 0.9 to 1 is due to the restriction on estimations of probability discussed in Section IV-A. Examining Figure 4 shows that PSP has the best with MCC at $\theta = 0.7$, and after that, the score decreases.

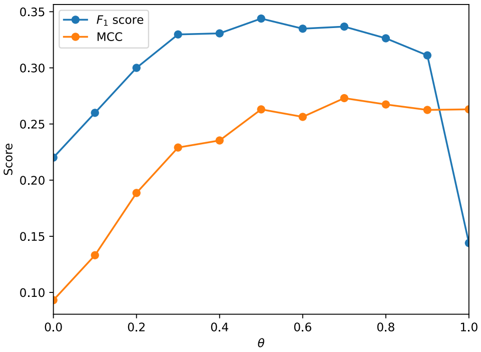


Fig. 4: This graph shows how PSP's performance varies with θ. Recall that, when $\theta = 0$, PSP conservatively considers all states deemed feasible under an abstract interpretation and, when $\theta = 1$, PSP is equivalent to the fuzzer it is using, and ignores all states the fuzzer did not produce. We see here that performance on F_1 score peaks at just shy of $\theta = 0.5$, whereas MCC peaks at around 0.7.

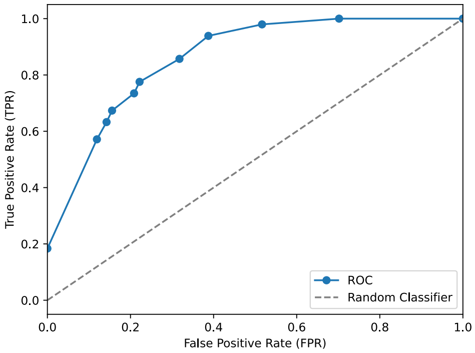


Fig. 5: This Receiver Operating Characteristic (ROC) curve shows how True Positive Rate and False Positive Rate vary with the threshold. We observe that PSP is strictly above a random classifier in terms of performance.

In Figure 5, we plot the Receiver Operating Characteristic (ROC) curve for $PSP(A_{30}, \theta)$ by considering only assertions visited by the fuzzer, as our PSP implementation cannot estimate assertions unvisited during fuzzing. A ROC curve shows the trade-off between the true positive rate and the false positive rate for different classification thresholds. The dotted

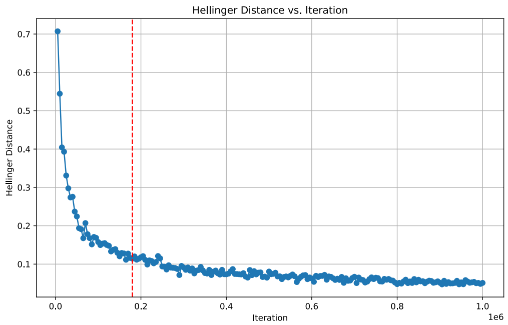


Fig. 6: This plot illustrates the average Hellinger distance between normalised PSPs across fuzzer iterations with a stride of 5000. The vertical dotted line indicates the number of fuzzer iterations corresponding to a 30 minute fuzzing campaign. We observe that the probabilities indeed converge when using PSP.

$x = y$ line represents the performance of a random classifier. The area between a classifier's ROC curve and the dotted line indicates how much the classifier outperforms random guessing. The fact that PSP's ROC curve is well above this line demonstrates that PSP is a decent bug classifier.

The fuzzing configuration A_{time} influences the quality of PSP. A_{time} depends on the fuzzer, which, in our case, is AFL. It also depends on how long the fuzzer runs to generate fuzzing data, which we measure in terms of time and the number of fuzzer iterations. In practice, a user of PSP may wish to identify what fuzzing budget is needed to construct a PSP of sufficient quality. We suggest doing it simply by observing the convergence of PSP as the budget increases.

To evaluate PSP's convergence, we computed the difference between PSPs across fuzzer iterations with the stride of 5000. To quantify the distance between the sets of state probabilities of two iterations, we normalise them, and then compute the average Hellinger distance [22] between the normalised probabilities as follows:

$$H(p_i, p_{i-5000}) = \frac{1}{\sqrt{2}} \sqrt{\sum_{i=1}^{n} \left(\sqrt{p_i} - \sqrt{p_{i-5000}}\right)^2}$$

where p_i denotes the probability at the i^{th} iteration.

Figure 6 shows how the distance between adjacent PSPs converges with the increase of the number of fuzzer iterations. The vertical dotted line indicates the number corresponding to a 30 minutes fuzzing campaign. The fact that PSP converges implies that we can stop the fuzzer after a certain number of iterations without a significant loss of precision.

E. Ablation: Estimating Unseen States

Table IV shows various estimating techniques for the unseen program state. We conducted fuzzing on Coreutils by randomly selecting lines for instrumentation, which we divided into two categories: dense and sparse. For the sparse category, we chose samples that did not satisfy the criteria for choosing the supervised model shown in Equation (3). We selected variables with a large range of values or samples from states that did not have enough fuzzing samples. For the dense category, we chose variables that satisfied the criteria for running the supervised model, such as error codes, file descriptors, enums, etc. This process resulted in 100k samples for sparse and dense data. Additionally, we generated discrete samples (DS) with specific patterns, such as even, odd, modulo, and other randomly generated periodic patterns. For string samples (SS), we extracted them by instrumenting strings from coreutils.

TABLE V: The performance of estimating techniques from Table IV. Coreutils-S denotes the sparse version of Coreutils, and Coreutils-D denotes its dense version.

Dataset	Samples	RFDIF	RFKDE	RF	DIF	KDE
Coreutils-S	100k	99.8%	99.5%	-	99.8%	99.5%
Coreutils-D	100k	99.3%	99.0%	98.4%	62.3%	56.4%

TABLE VI: Average patch ranking for MB35.

Bug	Rete	PSP-Rete
gmp-a1d3d-f17cb	843	840
libtiff-09e82-f2d98	513	496
libtiff-764db-2e42d	7	6
libtiff-a72cf-0a36d	15842	12039
libtiff-37133-865f7	8566	5711
php-70075-5a8c9	782	725
php-e65d3-1d984	5439	3954
php-63673-2adf5	3	3
Average	3999	2959

To evaluate the techniques, we considered any probability above 0.5 as indicating the existence of a state and any probability below 0.5 as indicating the non-existence of a state. Table V reports each technique's performance at estimating unseen states. RF is effective in recognising patterns present when the data is dense. However, for sparse data, both DIF and KDE are effective in estimating the unseen states, although DIF has slightly better results. For techniques such as symbolic execution on smart contracts, RF was rarely triggered (with a trigger rate of $< 1\%$) due to data sparsity. In shot, random forest (RF) outperforms the other methods on dense data; diffusing the probability (DIF) outperforms the other methods on sparse data.

F. Threats to Validity

We evaluated PSP against CLAM on a subset of Magma [9]. We used a uniformly sampled subset, of 7 out of 9, of the Magma benchmarks for computational reasons. The size of this sample set threatens the generalizability of our findings to other subjects. Further, PSP uses fuzzing to infer probabilities. A fuzzer, like AFL, may not cover certain variables or observe only a biased subset of a variable's bindings; this introduces another external validity threat. Our evaluation consists of running various tools (Mythril, AFL, Clam, Harvey). These tools are configurable; other than varying timeouts, we used their defaults. Here too, it is possible that our results do not generalise beyond these configurations.

VII. Related Work

PSP is relevant to data-driven program analysis, combinations of testing and verification, patch prioritisation for program repair, and symbolic execution.

Statistical Reasoning about Programs: This paper is most closely related with recent work on the development of empirical methods for program analysis, and specifically of statistical methods [23], [24], [25]. Empirical methods are particularly suitable for the analysis of large and complex systems where analytical (or formal) methods fail. Statistical methods allow to quantify the error and uncertainty in the analysis result. In this work, we explored advanced methods from machine learning, such as random forest and kernel density estimation, to estimate the probability that there exists an execution for which a given property is observed and discussed the integration with static analysis to tackle the problem of imprecision.

Data-Driven Program Analysis: Data-driven automatic tuning has been shown to improve static analysis performance [26], [27]. In contrast, we focus on the precision of program analysis. Heo *et al.* [28] use machine learning to balance the trade-off between precision and scalability of static analysis by selectively enabling unsoundness when analysing loops and library functions. Our technique, by relying on data obtained through fuzzing, directly approximates program states that enable a wider range of applications, including symbolic execution and program repair.

Combining Testing and Verification: Testing and verification have been combined in various ways [29]. Combining them to reduce the over-approximation of abstraction has been explored [30]. Tools such as SMASH [31] combine may and must analysis (over-approximation and under-approximation). UFO [32] uses interpolation to unify over and under-approximate techniques in model checking. PSP differs from these techniques by combining fuzzing data and abstract interpretation to construct probabilities for program states. This combination provides a trade-off between under- and over- approximation and reduces false positives.

Patch Prioritisation: DirectFix [33] uses weighted SAT to prioritise smaller changes. Prophet [34] learns a probabilistic model to rank patches based on maximum likelihood estimation. CapGen [35] estimates the likelihood of concrete patches using information from AST nodes. GetaFix [36] clusters mined fix patterns using a hierarchical clustering algorithm into general and specific fix patterns and uses code context to select an appropriate fix pattern. Trident [18] prioritises patches to reduce side effects and overfitting. Recent techniques such as L2S [37], ODS [38], and Rete [13] prioritise variables to reduce the number of candidate variables in the search space. Most techniques rank patches with scores that fall outside [0, 1]. We normalise their scores into [0, 1] so we can use Equation 11 to effectively combine them with PSP.

Symbolic Execution: Symbolic execution [39], [40] is one of the most expensive testing methodologies [41]. Much research seeks to reduce its cost [2]. Variants of symbolic execution have been proposed to reduce its cost, such as concolic execution [42], [43] and execution-based testing [44], [45]. Researchers studied ways to accelerate path exploration in symbolic execution. One such approach is distributing path exploration among different workers [46], [47]. Several techniques leverage compositionality to speed up symbolic execution [48], [31]. Researchers also investigated methods for pruning the search space [49], [50], and transforming the underlying code for symbolic execution [51], [52], [53].

One way to tackle symbolic execution's scalability challenge is to side-step it via search: rather than directly speed symbolic execution or solving, use search to maximise your computational budget. Kapus *et al.*'s Pending Constraints approach prioritises execution paths that are already known to be feasible and defers the rest [10]. Thus, like PSP, it avoids wasting resources on solving constraints. Pending Constraints is more conservative: it uses caching to identify *known-to-be-feasible* paths; PSP, in contrast, prioritises paths that it predicts will produce a likely state. Both approaches exemplify the "rich get richer" proverb: Pending Constraints will tend to explore feasible paths and their easy extensions more deeply, while PSP focuses on probable paths. For bug finding, we have shown PSP's prioritisation works better (Section VI-B).

Abstract symbolic execution (ASE) defines a value set decision procedure based on strided value interval sets for efficiently determining precise, or under-approximating value sets for variables, which helps to reduce the number of SMT queries. Furthermore, PSP approximates even further by reducing the number of queries as shown in Section VI-B.

Neuro-symbolic execution [54] trains a neural network to approximate hard-to-analyse program constructs such as loops and external function calls. In contrast, we directly approximate program states, which enables us to create an efficient search strategy reducing the number of SMT queries.

VIII. Conclusion

We present program state probability (PSP) to enhance program analyses such as abstract interpretation and symbolic execution, which suffer from imprecision due to over- and under-approximation. PSP uses execution samples to probabilistically approximate reachable program states and leverages these probabilities to increase analysis precision. We applied PSP to three domains: static analysis, symbolic execution, and program repair. Results show that using PSP reduces the number of false positives under abstract interpretation, increases the number of bugs that symbolic execution finds, and prioritises correct patches for program repair.

Acknowledgements

We thank Maria Christakis of TU Wien for her constructive feedback and Valentin Wüstholz of Consensys for his kind help modifying the Harvey fuzzer [7]. This work was enabled by the CIS @ Max Planck internship program and partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972.

References

[1] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.

[2] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[3] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[4] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 international symposium on software testing and analysis*, 2011, pp. 199–209.

[5] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[6] "American fuzzy lop (afl) - a security-oriented fuzzer." http://lcamtuf.coredump.cx/afl/technical_details.txt, accessed: May 5, 2023.

[7] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1398–1409.

[8] "Clam - static analyzer for llvm bitcode based on abstract interpretation." http://lcamtuf.coredump.cx/afl/technical_details.txt, accessed: May 5, 2023.

[9] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, Dec. 2020. [Online]. Available: https://doi.org/10.1145/3428334

[10] T. Kapus, F. Busse, and C. Cadar, "Pending constraints in symbolic execution for better exploration and seeding," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 115–126.

[11] A. S. Abyaneh and C. M. Kirsch, "Ase: A value set decision procedure for symbolic execution," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 203–214.

[12] "Mythril: Security analysis tool for evm bytecode," https://github.com/ConsenSys/mythril, accessed: May 5, 2023.

[13] N. Parasaram, E. T. Barr, and S. Mechtaev, "Rete: Learning namespace representation for program repair."

[14] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[15] P. Moin, *Fundamentals of engineering numerical analysis*. Cambridge University Press, 2010.

[16] W. Mendenhall, R. J. Beaver, and B. M. Beaver, *Introduction to probability and statistics*. Cengage Learning, 2012.

[17] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.

[18] N. Parasaram, E. T. Barr, and S. Mechtaev, "Trident: Controlling side effects in automated program repair," *IEEE Transactions on Software Engineering*, no. 01, pp. 1–1, 2021.

[19] Q. L. Le, A. Raad, J. Villard, J. Berdine, D. Dreyer, and P. W. O'Hearn, "Finding real bugs in big programs with incorrectness logic," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–27, 2022.

[20] J. Yao and M. Shepperd, "Assessing software defection prediction performance: Why using the matthews correlation coefficient matters," in *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, 2020, pp. 120–129.

[21] (2022) Introduction to smart contracts. [Online]. Available: https://ethereum.org/en/developers/docs/smart-contracts

[22] E. Hellinger, "Neue begründung der theorie quadratischer formen von unendlichvielen veränderlichen." *Journal für die reine und angewandte Mathematik*, vol. 1909, no. 136, pp. 210–271, 1909.

[23] M. Böhme, "Statistical reasoning about programs," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE 2022, 2022.

[24] ——, "STADS: Software testing as species discovery," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 2, pp. 7:1–7:52, Jun. 2018.

[25] S. Lee and M. Böhme, "Statistical reachability analysis," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023, 2023, p. 12.

[26] S. Jeong, M. Jeon, S. Cha, and H. Oh, "Data-driven context-sensitivity for points-to analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[27] M. Jeon, S. Jeong, S. Cha, and H. Oh, "A machine-learning algorithm with disjunctive model for data-driven program analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 41, no. 2, pp. 1–41, 2019.

[28] K. Heo, H. Oh, and K. Yi, "Machine-learning-guided selectively unsound static analysis," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 519–529.

[29] D. Beyer and M.-C. Jakobs, "Cooperative verifier-based testing with coveritest," *International Journal on Software Tools for Technology Transfer*, vol. 23, pp. 313–333, 2021.

[30] G. Yorsh, T. Ball, and M. Sagiv, "Testing, abstraction, theorem proving: better together!" in *Proceedings of the 2006 international symposium on Software testing and analysis*, 2006, pp. 145–156.

[31] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali, "Compositional may-must program analysis: unleashing the power of alternation," in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*, 2010, pp. 43–56.

[32] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "From under-approximations to over-approximations and back," in *Tools and Algorithms for the Construction and Analysis of Systems: 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24–April 1, 2012. Proceedings 18*. Springer, 2012, pp. 157–172.

[33] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 448–458.

[34] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[35] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1–11.

[36] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[37] Y. Xiong and B. Wang, "L2s: A framework for synthesizing the most probable program under a specification," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–45, 2022.

[38] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Transactions on Software Engineering*, 2021.

[39] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[40] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on software engineering*, no. 3, pp. 215–222, 1976.

[41] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 1066–1071.

[42] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[43] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[44] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[45] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Transactions on*

Information and System Security (TISSEC), vol. 12, no. 2, pp. 1–38, 2008.

[46] J. H. Siddiqui and S. Khurshid, "Scaling symbolic execution using ranged analysis," *ACM Sigplan Notices*, vol. 47, no. 10, pp. 523–536, 2012.

[47] M. Staats and C. Păsăreanu, "Parallel symbolic execution for structural test generation," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 183–194.

[48] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software: Practice and Experience*, vol. 30, no. 7, pp. 775–802, 2000.

[49] K. Taneja, T. Xie, N. Tillmann, and J. De Halleux, "express: guided path exploration for efficient regression test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 1–11.

[50] G. Yang, C. S. Păsăreanu, and S. Khurshid, "Memoized symbolic execution," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 144–154.

[51] J. Chen, W. Hu, L. Zhang, D. Hao, S. Khurshid, and L. Zhang, "Learning to accelerate symbolic execution via code transformation," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[52] J. Wagner, V. Kuznetsov, and G. Candea, "-overify: Optimizing programs for fast verification," in *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, no. CONF. USENIX Association, 2013.

[53] H. Converse, O. Olivo, and S. Khurshid, "Non-semantics-preserving transformations for higher-coverage test generation using symbolic execution," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 241–252.

[54] S. Shen, S. Shinde, S. Ramesh, A. Roychoudhury, and P. Saxena, "Neuro-symbolic execution: Augmenting symbolic execution with neural constraints," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/neuro-symbolic-execution-augmenting-symbolic-execution-with-neural-constraints/