SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning

Nicola Ruaro
ruaronicola@ucsb.edu
UC Santa Barbara

Lukas Dresel
lukas@ucsb.edu
UC Santa Barbara

Kyle Zeng
zengkyle@asu.edu
Arizona State University

Tiffany Bao
tbao@asu.edu
Arizona State University

Mario Polino
mario.polino@polimi.it
Politecnico di Milano

Andrea Continella
a.continella@utwente.nl
University of Twente

Stefano Zanero
stefano.zanero@polimi.it
Politecnico di Milano

Christopher Kruegel
chris@ucsb.edu
UC Santa Barbara

Giovanni Vigna
vigna@ucsb.edu
UC Santa Barbara

ABSTRACT

Exploring many execution paths in a binary program is essential to discover new vulnerabilities. Dynamic Symbolic Execution (DSE) is useful to trigger complex input conditions and enables an accurate exploration of a program while providing extensive crash replayability and semantic insights.

However, scaling this type of analysis to complex binaries is difficult. Current methods suffer from the path explosion problem, despite many attempts to mitigate this challenge (e.g., by merging paths when appropriate). Still, in general, this challenge is not yet surmounted, and most bugs discovered through such techniques are shallow.

We propose a novel approach to address the path explosion problem: A smart triaging system that leverages supervised machine learning techniques to replicate human expertise, leading to vulnerable path discovery. Our approach monitors the execution traces in vulnerable programs and extracts relevant features—register and memory accesses, function complexity, system calls—to guide the symbolic exploration. We train models to learn the patterns of vulnerable paths from the extracted features, and we leverage their predictions to discover interesting execution paths in new programs.

We implement our approach in a tool called SyML, and we evaluate it on the Cyber Grand Challenge (CGC) dataset—a well-known dataset of vulnerable programs—and on 3 real-world Linux binaries. We show that the knowledge collected from the analysis of vulnerable paths, without any explicit prior knowledge about vulnerability patterns, is transferrable to unseen binaries, and leads to outperforming prior work in path prioritization by triggering more, and different, unique vulnerabilities.

RAID '21, October 6–8, 2021, San Sebastian, Spain
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9058-3/21/10.
https://doi.org/10.1145/3471621.3471865

CCS CONCEPTS

• **Computing methodologies** → **Symbolic calculus algorithms**; **Supervised learning by classification**.

KEYWORDS

Symbolic execution, Vulnerability discovery, Machine learning

ACM Reference Format:
Nicola Ruaro, Lukas Dresel, Kyle Zeng, Tiffany Bao, Mario Polino, Andrea Continella, Stefano Zanero, Christopher Kruegel, and Giovanni Vigna. 2021. SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning. In *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21), October 6–8, 2021, San Sebastian, Spain.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3471621.3471865

1 INTRODUCTION

Automated vulnerability analysis systems are hard to develop and deploy in the real world; they tend to be guided by carefully balanced (and often not clearly understood) theoretical trade-offs to maintain feasibility. There are two main areas where such trade-offs must be made [31]:

- **Replayability:** *static* analyses create an offline (without execution) model of the application; they then apply heuristics and formal methods to find a potential vulnerability in a specific module, but cannot precisely trigger its execution. *Dynamic* analyses, instead, can execute the entire application, and, therefore, they can reason about the execution path that needs to be followed to trigger a vulnerability.
- **Semantic insight:** *concrete* analyses cannot reason about the program in semantically meaningful ways; they are not designed to understand which part of the input causes the application to behave in a specific way. On the other hand, a *symbolic* analysis can determine the specific portions of input responsible for certain program behaviors, resulting in a better semantic understanding.

Dynamic Symbolic Execution (DSE) is a promising approach for vulnerability discovery [4]. This technique executes the program in an emulated environment and in the abstract domain of symbolic variables, providing both semantic insight and replayability. As this approach emulates the application, it tracks the state of registers and memory. Whenever the execution hits a conditional branch and both conditions—the branch condition and its negation—are satisfiable, the execution forks and follows both paths, keeping track of the accumulated path constraints. DSE provides exceptionally high semantic insights into the target application, and can use the accumulated path constraints to produce an input that triggers a specific program state.

However, dynamic symbolic execution techniques suffer from very limited scalability due to the well-known *path explosion* problem. Since every branch results in the creation of new paths, the number of possible paths increases exponentially with the number of branch instructions (see *case 0* from Figure 1). Consequently, DSE has to limit the exploration to only a selected subset of execution paths, according to the amount of time and resources available.

There have been a number of attempts to mitigate the path explosion problem. *Symbolic-assisted fuzzing* offloads much of the processing to faster fuzzing techniques while retaining semantic insight [17, 33]. *Under-constrained Symbolic Execution* executes only parts of an application, giving up the replayability of detected bugs in exchange for scalability [16]. *Merging of execution paths* uses static symbolic execution techniques to over-approximate and merge different states when an appropriate condition is satisfied [3, 20, 30]. *Path prioritization* uses heuristics or traversal algorithms to find promising or less-explored paths [7, 21]. *Interleaved Symbolic Execution* provides the human analyst with an interface to combine concrete and symbolic execution [18].

However, all of the proposed techniques rely on specific metrics (e.g., edge coverage) to score and prioritize execution paths, which fail to take into account the patterns of paths that lead to vulnerabilities, and therefore do not capture the likelihood of the paths to exercise a bug. For this reason, such techniques often lead only to the discovery of few shallow vulnerable states, highlighting the need for more sophisticated and practical solutions to survive path explosion [33].

In this paper, we explore the effectiveness of novel path prioritization techniques, introducing a new approach based on *pattern discovery and learning*.

Prior work on path prioritization is broadly classified into two categories [23]. First, classic tree-traversal techniques such as Depth-First Search, Breadth-First Search, and Random Search are well-covered in the literature. They do not involve any knowledge of the program and are usually not practical when exploring complex programs. Second, heuristic-based techniques, such as KLEE's Coverage-Optimize [9], AEG's Loop Exhaustion [2], and Subpath-Guided Search [21], make use of specific metrics (e.g., coverage or loop behavior) to score and prioritize paths.

However, statically designed metrics are not generic enough. Prioritization heuristics often focus on a specific type of vulnerability: for example, coverage-based approaches are good at triggering logic vulnerabilities, but unlikely to find a memory corruption [1]. On the contrary, loop-based approaches are good at triggering memory corruption vulnerabilities, but unlikely to trigger a very specific

[1]Memory corruption vulnerabilities often require one to execute repeatedly a portion of the program. Research in the field of fuzz testing [39] shows that coverage can be an effective metric to trigger this type of vulnerabilities. However, this is not true for a coverage-driven pure DSE engine due to the speed disparity between fuzzing and concolic execution.

```
char username[100];
int admin = 0;

switch (choice) {
    case 0:
        // path explosion
        fgets(username, 100, stdin);
        for (int i = 0; i < 100; i++)
            if (username[i] == 'A')
                counter ++;
        break;
    case 1:
        // uninteresting code
        printf("Welcome!");
        break;
    case 2:
        // interesting code
        gets(username);
        if (authenticate(username) == 1) {
            admin = 1;
        }
        if (admin == 1) {
            privileged_function();
        }
}
```

Figure 1: A three-way switch statement. Only one of the branches contains an interesting function call, while the other branches are either uninteresting or are causing path explosion.

use-case. As a result, it is difficult for such approaches to detect different types of bugs.

Our Approach. In this paper, we propose a different approach, based on the extraction of features characterizing execution paths, functions, basic blocks, and connected components. By leveraging supervised machine learning techniques, we discover and learn the patterns of vulnerable paths. The robustness of this approach comes from its potential to gain insights into the nature of vulnerabilities, rather than providing possibly superficial heuristics to prioritize or merge execution paths.

The key intuition is that programs with similar bugs often have similarly dysfunctional execution states, characterized by a specific set of features—such as sequences of API calls, memory dereferences, very complex functions, or loop behaviors. This approach attempts to replicate a human analyst's expertise by recognizing paths that are more likely to lead to a security bug (see *case 1* and *case 2* from Figure 1).

We model our prioritization technique *without any explicit prior knowledge* about vulnerability patterns: we show that the knowledge gathered from the analysis of vulnerable traces is transferrable to unseen programs, and successfully guides the exploration to trigger a broad range of vulnerabilities, including some that went undetected by existing techniques.

We evaluate our technique on the Cyber Grand Challenge (CGC) dataset [32], composed of 232 distinct binaries with more than 400 distinct crashing inputs, showing that our approach can effectively prioritize promising paths, resulting in a broad range of triggered vulnerabilities. Our system, SyML, can identify and trigger 18 distinct vulnerable behaviors in the CGC programs under test, while the best technique from the related work can only trigger 11. SyML does not only trigger more vulnerabilities than any other technique, but it is also significantly less biased and can trigger 3 vulnerabilities that are undiscovered when combining results from all the previous techniques.

As a final step, we evaluate SyML on 3 real-world Linux CVEs, showing that the knowledge learned from the CGC dataset can be effectively transferred to unseen Linux binaries.

In summary, we make the following contributions:

- We propose a novel approach, based on supervised machine learning, for path prioritization in symbolic execution. Our primary intuition is that execution paths in programs that lead to similar bugs share similar, predictable features.
- We implement our approach in SyML, a system able to guide the symbolic exploration toward vulnerable states.
- We evaluate our approach on the CGC dataset (arguably one of the most robust datasets for automated analysis validation), showing that SyML outperforms prior work in path prioritization by triggering more unique vulnerabilities.
- We evaluate SyML on 3 real-world CVEs affecting Linux binaries, and we effectively transfer the models learned on the CGC dataset to achieve a better prediction accuracy.

In the spirit of open science, we make our code available at https://github.com/ucsb-seclab/syml.

2 PATTERN LEARNING

Intuitively, we expect legitimate programs to have a very low density of vulnerable paths, with more than 80% of the bugs found in less than 20% of the code [22]. Moreover, a vulnerable path is not only characterized by the last few executed blocks. It is often very context-dependent, meaning that only a very particular context (e.g., register and memory values) can trigger the vulnerability when executing the vulnerable code.

For this reason, it is the cumulative set of branching choices that ultimately distinguishes a good strategy from a bad strategy in DSE. Smart choices lead to interesting contexts and code regions; poor choices cause the analysis to wander toward uninteresting code.

We observe that distinct vulnerable code paths often contain similar programming patterns (e.g., specific input/output functions, specific sequences of instructions) and can be summarized and characterized using a finite number of features. Machine learning is well-suited to reason on existing data, figure out good indicators for vulnerable paths, and provide accurate predictions on unseen samples. We can then use such predictions to steer symbolic execution to more interesting contexts, eventually triggering more complex vulnerabilities.

This section presents the rationale behind our feature choice and the steps we performed to transform these features into reasonable numeric vectors. Considerations on the dataset, training, and validation infrastructure are left to Section 3.

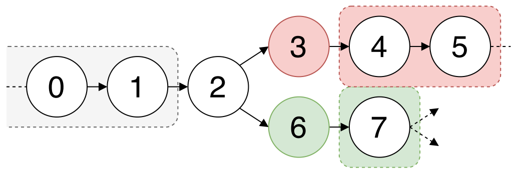

Figure 2: The feature extraction strategy leverages both the execution history and a *forward moving window* to inject context into the branching states. Each node represents a basic block in the CFG.

2.1 Feature Selection

As anticipated in the previous paragraphs, we consider the path prioritization problem as a branch prediction problem. Therefore, we represent the execution flow as a set of branching choices, where any branching state is associated with a set of features. Features can capture the current execution context while ignoring any program-specific noise, such as uninteresting fluctuations in register values or very complex operations. Table 1 presents a comprehensive list of the features that we use, with a brief description and motivation.

The feature extraction strategy leverages both the execution history and a *forward moving window* to inject context into the branching states: information relevant to such states is enriched considering up to `MAX_WSIZE` future states that belong to the branch [2].

Taking Figure 2 as an example, suppose a `MAX_WSIZE` of 2. After reaching state {2}, we pause symbolic execution and create two new training data points for states {3} and {6}.

The new data points are marked as `NON-TAKEN` or `TAKEN` **(F1)** to reflect which path was actually executed, and are loaded with local information (i.e., internal to the states) that summarize the branch's short-term behavior. Features **(F2)** to **(F5)** summarize the importance of the states in the program control flow, as well as the characteristics of the current function. Features **(F6)** and **(F7)** provide context about the control flow choices in the branch.

We then enrich both data points with historical information from the same past execution states {`0,1`} and parent state {`2`}. Feature **(F0)** indicates whether the branch was already visited, summarizing the exploration's past behavior.

Finally, we restart symbolic execution, and we enrich state {`3`} with forward information from window {`4,5`} and state {`6`} with forward information from window {`7`}. Features **(F8)** and **(F14)** represent the future and long-term behavior of the branch and its interactions with the environment.

The final value that we use for `MAX_WSIZE` is 2 states. The choice of this window size is driven by both accuracy and performance reasons. Table 2 presents the average exploration times and F1-score measures for different window sizes, ranging from 0 to 20 states. We obtain such measures from an analysis of the entire dataset—both the dataset and the analysis system are explained in detail in the following sections. Since the exploration times increase steadily with larger window sizes, and since we observe a peak in

[2] The information contained in the forward window is meant to provide a good summary of the branch behavior. The rationale behind this choice is to capture contextual information without necessarily explorating the entire branch, which would poorly affect performance.

Table 1: List of our features, together with their description and rationale. States refer to the example in Figure 2.

Feature	Description	Rationale
	PAST (i.e., states `{0,1}`)	
(F0) *num_branch_visits*	Number of times this branch was already visited.	It can be important for the exploration to repeatedly traverse an already visited branch (or explore a new one).
	BRANCHING STATE (i.e., states `{3}` and `{6}`)	
(F1) *taken*	Whether the branch is taken.	-
(F2) *connectivity*	Number of states connected in the CFG.	High connectivity indicates that the state is important for the program's control flow.
(F3) *centrality*	Centrality of the state in the CFG.	High centrality indicates that the state is important for the program's control flow.
(F4) *function_size*	Size of the current function.	Very large functions are a symptom of bad programming practices.
(F5) *function_complexity*	Cyclomatic complexity of the current function.	Very complex functions are a symptom of bad programming practices.
(F6) *leave_component*	Whether the branch is leaving the current component (e.g., a loop).	Depending on the context and functions involved, it can be important to either stay or leave the current component (e.g., to continue processing input, or allocate a new object).
(F7) *leave_community*	Whether the branch is leaving the current community (e.g., a set of similar functions).	Since each community represents a different logical subsystem in the program, it can be important to stay or leave the current subsystem.
	BRANCH (i.e., states `{4,5}` and `{7}`)	
(F8) *registers_read_write*	Number of register reads and writes.	Indicates how the program is interacting with the environment.
(F9) *memory_read_write*	Number of memory reads and writes.	Indicates how the program is interacting with the environment.
(F10) *address_concretizations*	Number of address concretizations.	Indicates how the program is interacting with the environment.
(F11) *num_calls*	Number of function calls that appear in this branch.	Indicates how the exploration is interacting with the program.
(F12) *num_returns*	Number of return statements that appear in this branch.	Together with the number of calls, it indicates how the exploration is interacting with the program.
(F13) *num_syscalls*	Number and type of system calls that appear in this branch.	Can be leveraged together with context information to trigger specific program behaviors.
(F14) *num_communities*	Number of communities that are traversed in this branch.	Indicates how the exploration is moving between different logical subsystems in the program.

Table 2: Average window exploration times and F1-Score measures for multiple window sizes.

Window Size	**0**	**2**	**5**	**10**	**20**
Exploration Time	0.01s	0.23s	0.38s	0.57s	0.77s
F1-Score (RandomForest)	92%	92%	86%	83%	82%
F1-Score (AdaBoost)	92%	93%	86%	83%	83%
F1-Score (XGBoost)	91%	94%	89%	86%	85%

the accuracy of the models with a window of size 2, we decide to use a `MAX_WSIZE` of 2 for the following experiments.

2.2 Feature Preparation

Algorithms do not learn as humans do. Computers cannot directly reason upon natural language, words, and complex data structures. This section describes the non-obvious assumptions and transformations used to represent all our features conveniently.

Our feature set contains two main types of complex data: *communities* and *syscalls*.

Communities represent a logical partitioning of connected subsystems in the target program's call graph. Figure 3 presents an example of the communities for the program `CADET_00001`. Each community is a subset of the program's functions that maximizes modularity (connections within the same community are dense, while connections between different communities are sparse). Specifically, we compute the community partitions using the Louvain algorithm [6]. As a result, communities will cluster together functions that are well interconnected and have a strong relationship in the call graph.

Communities are challenging to represent as they are different for each program and, therefore, they are not easily learned by our models. We decide to summarize communities with two distinct

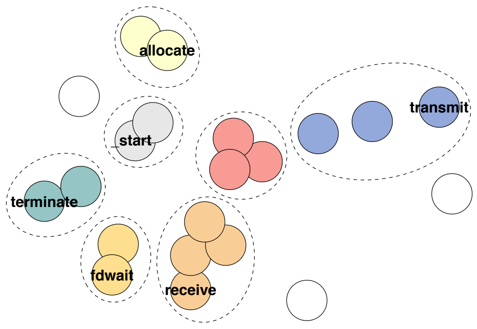

Figure 3: Examples of Louvain communities partitions for the program `CADET_00001`, where each node represents a function in the program. Functions with a strong relationship in the program's call graph are clustered together.

features, *num_communities* represents the number of traversed communities in the current branch, and *leave_community* indicates whether the exploration moves from one logical subsystem to another.

Similarly, we represent *syscalls* with the feature *num_syscalls*, which indicates the total number of syscalls in the current branch.

3 MODEL PREPARATION

3.1 Dataset

The dataset we choose for training and validation is the Cyber Grand Challenge (CGC) dataset [32], composed of 232 vulnerable programs with more than 400 distinct crashing inputs triggering a wide range of vulnerabilities. In particular, due to the shortcomings of our DSE engine and symbolic re-tracing framework (see Section 6), some of the vulnerabilities cannot be analyzed and are not in our training set. Specifically, 29 of the binaries could not run in our DSE engine, and desynchronizations in the re-tracing framework limit our training set to 120 vulnerabilities across 75 binaries. Such inaccuracies in the DSE engine affect all the binaries and vulnerability classes equally. Therefore, the distribution of vulnerabilities in our training set remains unaffected. It is out of the scope of this paper to address such issues, which are described in more detail in Section 6. Nonetheless, we do not restrict the exploration to the binaries in our training set, and we validate every technique (including SyML) against the entire dataset.

To select a good dataset, we took into account several aspects, which motivate our choice. **Volume:** it is empirically known [15] that the volume of available data is crucial to machine learning since it directly allows us to tackle more ambitious problems. **Variety:** learning from various binaries, subject to many different classes of vulnerabilities, allows models to generalize. **Consistency:** binaries compiled from different languages, different architectures, or with different compilers result in a noisy dataset. **Complexity:** simple binaries individually crafted to be vulnerable would result in overfitting the dataset. **Confidence:** we should have a solid knowledge of every vulnerability present in the binaries, along with relevant crashing inputs.

On these premises, the CGC corpus, along with crashing inputs and well-documented vulnerabilities, creates a robust training set. First, the challenge binaries are complex programs such as games, content management systems, and image processors [1]. Indeed, to be effective, analysis tools must process software with a low bug density. Second, instead of injecting additional synthetic bugs into existing programs [14], every vulnerability in this corpus is designed and documented with high confidence, creating a reliable ground truth for our algorithms.

3.2 Feature Extraction

Feature extraction is where we transform binaries and crashing inputs into a relevant set of features. The high-level process consists of three steps: concrete tracing, static analysis, and Dynamic-Symbolic tracing.

Concrete Tracing. First, we run all binaries in the QEMU [5] emulator and test them against the set of crashing inputs. This results in a collection of multiple execution traces. The execution traces contain only a list of the executed basic blocks, as opposed to other concrete tracing approaches [23] that also keep track of nondeterministic events during the execution.

Static Analysis. We then collect static global information, such as the Control Flow Graph (CFG), to support future analyses. We align the basic blocks in the CFG (i.e., normalization) to match the execution blocks used by the QEMU emulator and guarantee that there is no overlap between any two basic blocks.

Dynamic-Symbolic Tracing. Finally, we symbolically execute the crashing binaries using *angr*'s `Tracer` exploration technique [31], which forces the execution to follow the recorded trace. Symbolic execution is virtually interrupted at each fork (i.e., when reaching a branch), and a new data point is created for every resulting branch.

3.3 Cleaning the Data

During *Dynamic-Symbolic Tracing*, we extract and prepare our features. After creating all the training data points, we systematically clean the dataset from missing values, outliers, anomalous values, and duplicates.

Missing values, such as `NaN` and `Null`, are replaced with *zeroes* as they are associated with either a block with no successors (i.e., a *leaf node* in the CFG) or an empty block (i.e., *syscalls*).

Outliers and infinite values are adjusted to lie in the `2nd to 98th` percentile range to enforce more stable models.

Duplicated data points are removed from the dataset. Moreover, because of the limited feature set, some execution states are different but indistinguishable in practice. In particular, some loop iterations result in two identical data points except for the taken field. Since it is improbable—and undesirable—for the models to learn to follow a loop for a precise number of times, this is handled by ignoring the exit branch. The models' priority remains to find vulnerable program regions, while we defer the control over loop iterations to the prioritization framework presented in Section 4.

As a final step, we normalize all features with respect to the forward window size (`MAX_WSIZE`).

3.4 Training

We frame the prediction problem as a supervised classification problem. We aggregate all the branch decisions observed during the analysis to create our training set, and we train each model to predict whether or not a branch should be taken to reach a vulnerability. The metrics that we consider to evaluate our classification models are *F1-Score*, *Accuracy*, *Coverage*, and *Time-to-Score*—i.e., the time spent generating a prediction.

F1-Score, denoted by F_1, is the weighted average of *Precision* (the fraction of relevant blocks among the retrieved blocks) and *Recall* (the fraction of the total amount of relevant blocks retrieved). We use this metric to evaluate our classification models as it is widely used in the literature.

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Accuracy is the number of correct predictions over the total number of training data points. More accurate models result in better decisions; this makes accuracy one of the standard metrics in machine learning.

$$Accuracy = \frac{\#\ Correct\ Predictions}{\#\ Data\ Points}$$

Trace Coverage is a non-standard metric representing the fraction of blocks in the execution trace that we predict at least one time correctly. Basic blocks are often executed multiple times, either in a loop or as the result of repeated function calls. Therefore, the correct prediction of one frequent branch can have a high impact on the measured accuracy. Our intuition is that we want the models to be accurate *across the binary* and traverse it without critical prediction errors. Correct predictions must be spread across all functions, eventually covering most of the basic blocks in the execution trace.

$$TraceCoverage = \frac{Distinct(Correct\ Predictions)}{Distinct(Execution\ Trace)}$$

We evaluate the classification models using cross-validation at the granularity of individual binaries. During each cross-validation round, one single binary is kept out from the training set, regardless of the number of vulnerabilities it contains. As a result, the models cannot learn any pattern from other vulnerabilities in the left-out binary, which reduces biases. We then train and validate each model against each binary, extracting F1-Score, Accuracy, Coverage, and Time-to-Score measures. The iteration of this process for n cross-validation rounds, with n equal to the number of binaries, prevents the models from overfitting and assesses their ability to generalize over different binaries.

4 GUIDED SYMBOLIC EXECUTION

This section presents the strategies, optimizations, and overall framework that we use to prioritize paths and explore vulnerable programs by efficiently leveraging the trained models.

Figure 4 presents the information flow for our overall approach. In previous sections, we described that vulnerable binaries and crashing inputs are executed concretely to determine the execution traces. We then emulate these execution traces in the symbolic domain and extract a list of features—the dataset. After preparing the dataset (Section 3.3), we train a group of supervised learning models, and we leverage the predictions of the learned models to obtain scores and guide the symbolic execution.

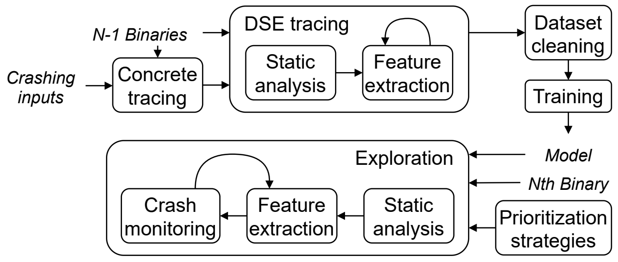

Figure 4: Information flow for the overall approach.

However, predictions are not perfect, and the search space is often large. Therefore, it is necessary to employ a resilient and effective strategy, make the best possible use of predictions, and reliably discriminate between interesting and uninteresting paths.

4.1 Performance Considerations

Extracting features and retrieving a prediction (either `TAKEN` or `NON-TAKEN`) should not cause a dramatic slow-down in the analysis. However, depending on the number of features, the complexity of features, and the complexity of models, feature extraction time and time-to-score can grow and become the bottleneck for the analysis. These considerations led us to two main trade-offs between scalability and performance.

Initial overhead. To avoid excessive overhead during the exploration, we pre-compute features independent from the context as an initial overhead, and we re-use them whenever they are needed by the analysis. Features that are initially computed are *connectivity*, *centrality*, *function size*, *function complexity*, *components information*, and *communities partitions*.

Number of features. Having a high number of features is harmful to both computation time and model accuracy [15]. For this reason, we use *information gain* to manually inspect all our features and ensure that they are both *non-redundant* and *relevant*. As a result, every feature in the final feature set (see Table 1) contributes to discriminate `TAKEN` and `NON-TAKEN` branches. For example, features that we found irrelevant are *syscalls_terminate* and *syscalls_random*. These features are ignored for both the training and the exploration stages, resulting in both more efficient feature extraction and more effective models that provide quick predictions.

4.2 Exploration Technique

During the analysis, we represent the execution with three mutable path **stashes**: `active`, `deferred`, and `crashed`. The stash is the structure that we use to store and organize the execution states, and any state resulting from symbolic execution is stored in one of these three stashes.

(1) States are put in the `active` stash if and only if they are considered good enough to be prioritized. Only one state at a time can be in the `active` stash.

```
input: program, model, strategy
output: crashing_input

begin
  simgr ⟵ new simulation_manager(program)
  static_info ⟵ do_static_analysis(program)
  featurizer ⟵ new featurizer(static_info)

  while len(simgr.active) > 0
    simgr.step()

    for s in simgr.active
      if is_crashed(s)
        simgr.crashed ⟵ s
        return s.posix.stdin

    if len(simgr.active) > 1
      // this is a branch
      features ⟵ featurizer.get_features(simgr.active)
      scores ⟵ model.score(features)
      update_scores(simgr.active, scores)

      simgr.deferred ⟵ simgr.active
      simgr.active ⟵ strategy.choose(simgr.deferred)
end
```

Figure 5: Pseudocode of the exploration technique.

(2) States are put in the `deferred` stash if they are not the current choice but may be chosen shortly. Due to memory constraints, we set a size threshold for the `deferred` stash. When the number of states gets over this threshold, we discard them according to the prioritization strategy.

(3) States are put in the `crashed` stash if and only if they are causing the program to crash.

Figure 5 presents an overview of how we implement the exploration technique in practice. Initially, our system creates a `simulation manager` object. The simulation manager is a control interface used by the *angr* DSE engine to handle multiple program states and stashes simultaneously. We store the program's static information, such as the CFG, in the `static info` object, and we use it to initialize the `featurizer` object. During the symbolic exploration, whenever we hit a new branch, we use the `featurizer` object to extract its features and update its score based on the model's prediction. We store all the potentially interesting branches in the `deferred` stash. After each execution step, we apply the provided prioritization `strategy` to select and move the most interesting execution state to the `active` stash. When an execution state crashes, we store it in the `crashed` stash and return its concretized input.

The exploration technique is, in summary, a set of rules which rigorously define how to move the states between the `active`, `deferred`, and `crashed` stashes.

4.3 Prioritization Strategies

During the feature extraction stage, as described in Section 3.2, every branch leads to the creation of a new candidate path. We refer to the set of candidate paths as the `deferred` stash, and prioritization strategies define an *order relation* over this set.

The strategy used for prioritization is at least as important as the predictions themselves. Spotting false positives is crucial to avoid path explosion and not to mislead the analysis. Spotting false negatives, on the other hand, is crucial to avoid de-prioritizing vulnerable paths.

Moving stats. Our system always uses scores with a per-path granularity. The score of each branching choice is a floating-point number and coincides with the model prediction. However, when referring to a particular path, we refer to the combined score of its branching choices, which is updated whenever a new step affects the path in question. In particular, the statistic that shows the best results is a moving average with a small sliding window (*N=2*).

$$\overline{score}(p) = \frac{1}{N}\sum_{i=0}^{N} score_{p,\ n\_branches-i} \qquad \forall\, p \in deferred$$

The choice of a small sliding window gives more importance to new branches and helps in prioritizing paths with good scores without over or under-prioritizing paths after a single wrong prediction.

Our approach uses two different prioritization strategies.

FAST Strategy. This strategy recursively queries the models, treating scores as an absolute value and prioritizing only the top-scored path.

$$next = \underset{p \in deferred}{\arg\max}\ \{\overline{score}(p)\}$$

BALANCED Strategy. Sometimes paths are very similar, and having more than one promising path is possible. Therefore, this strategy uses scores as a probability measure.

$$\Pr(next = p) = \overline{score}(p) \qquad \forall\, p \in deferred$$

While the `FAST` strategy would force the analysis to pick only the highest score, the `BALANCED` strategy makes a weighted choice to pick one path among the most promising ones.

Similarly to what KLEE [9] does, we interleave the two strategies to improve the overall effectiveness and protect against cases where an individual strategy gets stuck.

5 EVALUATION

We evaluated SyML and compared it against the state-of-the-art approaches in path prioritization by integrating and running each technique in the *angr* framework. In particular, we compared against KLEE Random search [9], KLEE Coverage Optimize search [9], and AEG Loop Exhaustion search [2]. Additionally, we performed a detailed analysis of the scores that SyML is able to produce.

5.1 Experimental Setup

Before taking any further steps into the analysis, we must appropriately set up the execution environment. More precisely, we initialize the DSE environment and enable the options to force a strict page access policy and `NX` memory protection as a heuristic to monitor the execution and detect crashing states. Furthermore, we use QEMU to trace suspicious states that emerged during the execution (e.g., unconstrained, dead-ended) and identify additional undetected crashes.

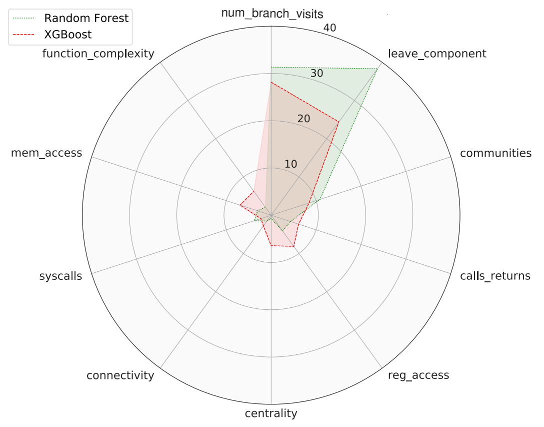

Figure 6: Feature importance measurements for the XGBoost and Random Forest models, reflecting the percentage score variations induced by each feature. Higher values indicate that the feature is more influential in the predictions of the model.

As discussed in Section 3.3, the control over loop iterations is left to this framework and explicitly handled using a *hard limit on consecutive loop iterations*. We perform static analyses as a fixed initial overhead, and we collect the CFG along with static information about functions (e.g., size, complexity, syscalls) and other components (e.g., loops, communities). The information collected from these analyses is used throughout the execution as the input for feature extraction.

We run our experiments on an Ubuntu 18.04 system with a maximum clock frequency of 3.6GHz and a memory limit of 8GB of RAM per process. We assign one CPU core for the analysis of each binary. Finally, we run and monitor all the techniques for 24 hours.

5.2 Model Accuracy and Feature Evaluation

We train and validate multiple machine learning algorithms: *Logistic Regression*, *Linear Discriminant Analysis*, *K-Nearest Neighbors*, *Support Vector Machines*, *Multi-Layer Perceptrons*, *Decision Trees*, *Random Forests*, *AdaBoost*, and *XGBoost*. In particular, all models are instantiated using the `scikit-learn` machine learning framework [28]. As described in Section 3.4, we evaluate the classification models using cross-validation. During each cross-validation round, one single binary is kept out from the training set, regardless of the number of vulnerabilities it contains. We present the results from the cross-validation in Table 3.

Table 3: Average values for the F1-Score, Accuracy, Trace Coverage, and Time-to-Score metrics after multiple rounds of cross-validation on CGC binaries.

Model	F1	Accuracy	Trace Coverage	Time-to-Score
`LogRegr`	77%	66%	73%	0.01s
`LinDiscr`	76%	68%	75%	0.01s
`KNN`	79%	63%	70%	0.1s
`SVM`	82%	76%	72%	0.04s
`MLP`	81%	80%	68%	0.04s
`DecisionTree`	85%	80%	78%	0.02s
`RandomForest`	92%	90%	90%	0.32s
`AdaBoost`	93%	91%	83%	0.02s
`XGBoost`	94%	93%	91%	0.2s

The Random Forest and XGBoost models achieve the best results on our dataset in terms of both *Accuracy* and *Trace Coverage*. The *Accuracy* scores are 90% for the Random Forest model and 93% for the XGBoost model. The *Trace Coverage* scores are 90% and 91%, respectively. We associate higher *Accuracy* scores with better decisions, as described in Section 3.4. Similarly, we associate higher *Trace Coverage* scores with greater robustness of the model predictions. In particular, the *Trace Coverage* of Random Forest and XGBoost is much higher than the rest of the models, indicating that both are robust and well-suited to guide symbolic execution.

These models are also relatively simple and highly interpretable. We evaluate the importance of different features by computing their permutation importance [8], an estimation method based on the score variation induced by each feature. Figure 6 presents the feature importance measures for the XGBoost and Random Forest models.s We observe that not all features are equally important. Features that contribute the most to the classifiers' predictions are the `number of branch visits`, `communities`, and `register access`, and `leave component`. Other features, such as `syscalls`, are influential for other models but do not contribute much to these models' splitting conditions.

Finally, hyper-parameters for the best model are tuned using a validation set. We use an initial *random search* to explore possible configurations due to the high-dimensionality of the parameter space. We then exhaustively search the restricted parameter space to ensure the most desirable configuration.

The main hyper-parameters that we tune for the Random Forest model are: the data points sampling method used for each tree (`bootstrap`), the maximum number of trees in the forest (`num_estimators`), the maximum depth of each tree (`max_depth`), and the maximum number of features considered to split a node (`max_features`). Similarly, the hyper-parameters tuned for the XGBoost model are: `bootstrap`, `num_estimators`, `max_depth`, and `min_child_weight`. Most of these hyper-parameters are described in the previous paragraphs. Additionally, `min_child_weight` expresses a limit on the number of data points that each node must contain.

We select the XGBoost model as our final model because of its better *accuracy* and shorter *time-to-score*. The resulting model uses `150` estimators, with the maximum depth of each tree limited to 2, and a minimum number of 6 data points in each node. For a more in-depth description of each parameter, please refer to the `scikit-learn` [28] documentation.

5.3 Comparison with Existing Techniques

We compare our approach with state-of-the-art techniques in path prioritization. To avoid introducing bias in our comparison, we implement all the path prioritization techniques in the same framework (i.e., *angr*). This allows us to exclude external factors (e.g., degree of formula approximation, level of preprocessing, implementation of the symbolic function stubs) that are intrinsic to the symbolic execution framework. We implement each technique referring precisely to both the authors' description and its reference implementation, if publicly available [9].

All the techniques are appropriately initialized and run without interruptions, as explained in Section 5.1. Moreover, we repeat the experiment two times and consider the average time to crash for all the vulnerabilities triggered in both repetitions.

KLEE Coverage Optimize [9]: this strategy attempts to select states likely to cover new code in the immediate future. The heuristic used to compute a weight for each path combines the *minimum distance to an uncovered instruction* and whether the path has *recently covered any new code.*

KLEE Random [9]: this strategy is conceptually similar to a randomized breadth-first search. It selects a path by traversing the execution tree from the root and randomly selecting the branch to follow at branch points. Therefore, when reaching a branching point, each subtree's set of paths will have an equal probability of being selected, regardless of their size.

AEG Loop Exhaustion [2]: this strategy builds up from the intuition that paths involving more loop iterations are more promising to produce memory corruption bugs, such as buffer overflows. Whenever execution hits a symbolic loop, the loop is executed as many times as possible until it is exhausted.

We present the results of our evaluation in Table 4 and Figure 7. Table 4 presents a comprehensive report of our results in terms of time-to-crash, tagged according to the class of vulnerability triggered. Overall, we triggered 24 distinct vulnerable states in 22 binaries. Figure 7 presents the progress (i.e., the cumulative number of vulnerabilities found) of each technique during the 24 hours of exploration. The exploration is not interrupted after the first crash, allowing the techniques to trigger more than one vulnerability in the same binary (see `NRFIN_00016`, `NRFIN_00023`). Moreover, we match each crash address and stack trace with the vulnerabilities documented in the CGC dataset. Our search strategy, SyML, triggered 18 distinct vulnerabilities in 16 binaries, 6 more than the best

Table 4: Time-to-Crash and vulnerability type for all the vulnerabilities found. `SOF`: Stack Buffer OverFlow, `HOF`: Heap Buffer OverFlow, `OBR`: Out-of-Bound Read, `OBW`: Out-of-Bound Write, `INT`: Integer Overflow, `TYPE`: Type Confusion, `PTR`: Untrusted Pointer Dereference, `FMT`: Format String. ✗ indicates that the technique did not trigger any vulnerability.

Binary	Vulnerability Type	KLEE Coverage	KLEE Random	AEG Loop Exhaustion	SyML
CADET_00001	SOF	✗	✗	2m	5m
CADET_00003	SOF	✗	✗	2m	6m
CROMU_00019	SOF	✗	✗	1h 8m	11h 16m
EAGLE_00005	SOF	✗	✗	2h 44m	✗
NRFIN_00016	SOF	1h 28m	1h 28m	12m	1h 16m
NRFIN_00016	SOF	✗	✗	1h 46m	14h 44m
NRFIN_00023	SOF	✗	✗	✗	3h 13m
YAN01_00001	SOF	✗	✗	16m	17h 13m
YAN01_00016	SOF	✗	✗	1m	4m
CROMU_00006	HOF	✗	2h 21m	✗	1h 22m
CROMU_00014	HOF	1h 4m	✗	✗	✗
KPRCA_00057	HOF	✗	✗	✗	15h 41m
YAN01_00012	HOF	✗	✗	19m	✗
CROMU_00012	OBW	✗	✗	2m	18h 11m
CROMU_00036	OBW	5h 50m	3m	✗	40m
CROMU_00034	OBR	43m	1h 35m	✗	2h 55m
NRFIN_00052	INT	32m	3h 7m	✗	5h 46m
KPRCA_00014	INT	✗	54m	✗	2h 8m
KPRCA_00033	TYPE	12m	2h 14m	47m	✗
KPRCA_00015	PTR	4m	35m	✗	✗
NRFIN_00023	PTR	2m	1m	✗	3m
NRFIN_00039	PTR	✗	✗	✗	9h 6m
CROMU_00043	FMT	1h 29m	✗	✗	4h 31m
KPRCA_00038	FMT	2h 59m	14h 31m	✗	✗
DISTINCT VULNERABILITIES	*(24)*	10	10	11	18
DISTINCT BINARIES	*(22)*	10	10	10	16

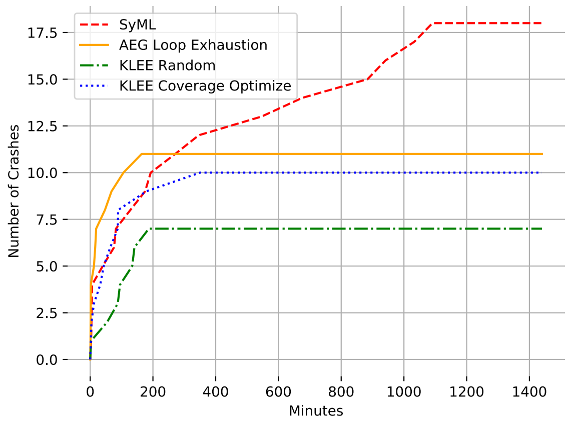

Figure 7: Crashes [Number] over Time [Seconds]. Cumulative number of vulnerabilities found during the exploration.

technique from the state-of-the-art—AEG Loop Exhaustion triggered 11 crashes in 10 binaries. Our results highlight the limitations of all current approaches. Heuristics primarily target one specific category of bugs, and this introduces a strong bias in the analysis.

The KLEE Coverage Optimize strategy tends to trigger edge-case vulnerabilities. It is effective in finding bugs that derive from inadequate testing of the application. However, it is not effective in crafting specific contexts to trigger more deep vulnerabilities.

The KLEE Random strategy performs a random exploration of the search space. Therefore, it unreliably triggers a wide range of shallow vulnerabilities, struggling to follow loops or specific execution patterns.

The AEG Loop Exhaustion blindly exhausts all the symbolic loops. Thus, it is exceptionally reliable in finding overflows and repetitive patterns, while missing almost any other type of bug.

SyML can trigger more vulnerabilities than any other technique, and triggered most of the vulnerabilities found by other strategies without showing any bias toward specific classes of bugs. While each of the techniques from the state-of-the-art has its apparent strengths and shortcomings, SyML does not show any obvious shortcomings. Our system can find vulnerabilities ranging from `Stack Buffer Overflows` to `Untrusted Pointer Dereferences` and `Out-of-Bound Reads`. In our experiments, it only missed the `Type Confusion` vulnerability class (see `KPRCA_00033`). Moreover, SyML can trigger vulnerabilities not found by any other technique (see `NRFIN_00023`, `NRFIN_00039`, `KPRCA_00057` from Figure 4), indicating that its prioritization strategy is effective and improves prior work results.

We notice that many vulnerabilities are still not triggered by any technique and that most of the vulnerabilities emerge during the first 12 hours of exploration, emphasizing that our system mitigates the path explosion problem effectively, but not completely.

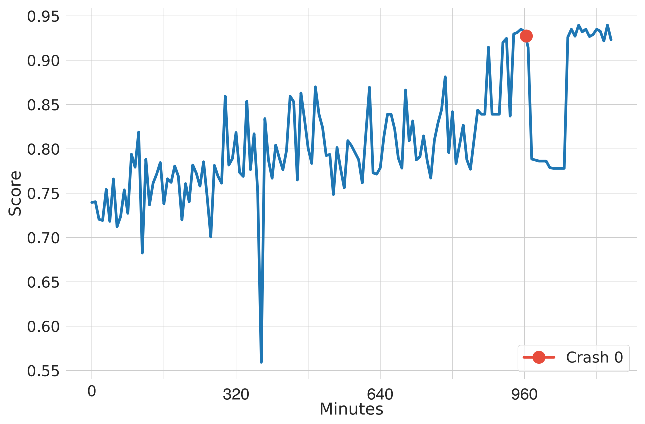

Figure 8: Score [Decimal Percent] over Time [Minutes]. Example of scores distribution for the program `CROMU_00012`.

5.4 Score Analysis

We analyze SyML score distribution during the exploration. In general, we expect the scores to increase when approaching a vulnerability and decrease otherwise. Our analyses confirm this expectation. We find that the scores assigned by our system, SyML, are consistent with the resulting crash patterns. As an example, Figure 8 presents the distribution of the scores from the exploration of the `CROMU_00012` program—we observe similar exploration patterns in many of the programs analyzed. Initially, scores are relatively low and unstable. We observe a score increase when approaching the vulnerability, indicating that the system is increasingly interested in the execution path. Scores reach a plateau coinciding with the crashing point, and after a sharp drop, they are moderately stable at the plateau point until the analysis is interrupted. Even after the crash, scores remain high (after a short drop), indicating that the system is still exploring an area close to the crashing point or finds the execution path still moderately interesting.

5.5 Applications to real-world Software

We evaluate our approach on 3 real-world Linux binaries: `asp2php` (`CVE-2004-1261`), `o3read` (`CVE-2004-1288`), and `ringtonetools` (`CVE-2004-1292`). The 3 vulnerabilities consist of a buffer overflow in the parsing module of each binary.

We run SyML on the 3 CVEs to extract the features and prepare the dataset, as described in the CGC experiments. After this initial

Table 5: Average F1-Score and Accuracy after multiple rounds of cross-validation on Linux binaries.

Model	F1	Accuracy
`RandomForest (Linux)`	63%	70%
`AdaBoost (Linux)`	63%	63%
`XGBoost (Linux)`	51%	56%
`XGBoost (CGC)`	69%	54%
`XGBoost (CGC+Linux)`	77%	66%

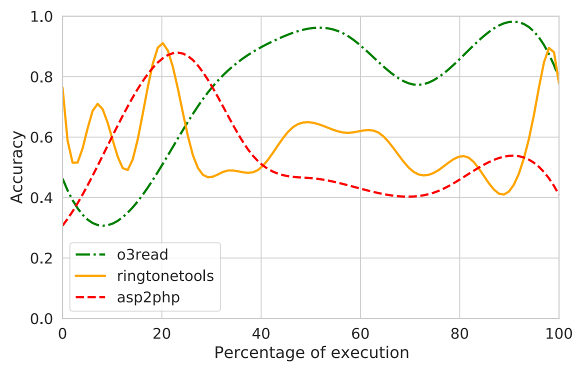

Figure 9: Spline curves approximating the prediction accuracy [Decimal Percent] at different program points [Percent] for the 3 analyzed Linux CVEs. Program points at 0% are close to the start of the execution, program points at 100% are close to where the vulnerability is triggered.

step, we train and cross-validate different machine learning algorithms. Table 5 presents our results after the cross-validation. The model that achieves the best performance is the `Random Forest` model, with an accuracy of 70% and an F1 score of 63%.

Transfer Learning: The accuracy and F1 scores presented in Table 5 are still not comparable with those achieved on the larger CGC dataset. For this reason, we re-use the `XGBoost` model from the CGC experiments to transfer its knowledge to this new dataset. The accuracy obtained with such model, without any re-training, is low. However, after updating the model's internal state to consider the new Linux dataset, we obtain an accuracy of 66% and an F1 score of 77%. Thus, the combined model achieves the best performance, showing promising results regarding the knowledge transfer between the CGC dataset and the Linux dataset.

Prediction Analysis: We evaluate the accuracy of the model's predictions at different points in the execution trace. Figure 9 presents our results. We observe higher overall accuracy in 1 of the 3 binaries (`o3read`), and in general, high accuracy in the proximity of the vulnerability. We believe that these results are important for two main reasons. First, the accuracy achieved by our models shows that we can effectively train a machine learning model that can predict the vulnerable paths in a Linux program with good accuracy. Second, we show that the results discussed in the previous sections are not restricted to the CGC dataset. In fact, the semantics of CGC binaries are analogous to the Linux x86 semantics, and this allows us to transfer some of the knowledge learned from the larger CGC dataset to the Linux dataset.

6 DISCUSSION

Our experiments show that SyML introduces a novel prioritization technique that can mitigate the path explosion problem by learning interesting patterns from vulnerable execution paths.

Even though this work is merely the first step toward more effective vulnerability discovery techniques, for the first time a machine learning approach is successfully used to discern and rank interesting branches in a DSE engine. To the best of our knowledge, our solution is the most effective way to confront the problem. However, the main limitation of this work is the lack of a large-scale experiment on Linux software. It is significantly harder to emulate a program's execution in the symbolic domain than in the concrete domain. Partial execution traces (without environment knowledge) introduce inaccuracies and eventually cause the tracing process to desynchronize, resulting in an incomplete dataset. Similarly, inaccuracies introduced by the DSE engine can limit the exploration and prevent the system from triggering vulnerable states. For these reasons, and considering the current state of pure DSE systems, it seems unfeasible at the moment to design such a large-scale experiment with a similarly sized dataset running entirely on Linux software.

The CGC programs are statically compiled x86 binaries, with semantics equivalent to Linux binaries but running on a different OS with a smaller set of system calls—DECREE. Since DSE handles very well the simplified environment, we decided to evaluate our approach on CGC binaries to sidestep these technical problems unrelated to our core contribution. Nonetheless, if DSE inaccuracies were not a concern, and with the right dataset, we believe that our system, SyML, would be flexible enough to run on Linux software.

Future work. Pure DSE engines are still fundamentally limited, and trying to analyze more complex programs reveals several problems. The technique here described could be adapted and applied to guide a Hybrid Fuzzing engine, which is in general more performant than a pure DSE engine, and we believe would be another interesting research direction [10, 11]. Moreover, using a different re-tracing framework [25] can help mitigate the problem of partial execution traces, reducing the impact of desynchronizations and allowing the concolic tracer to scale to large Linux programs.

7 RELATED WORK

Path explosion and coverage of the program search space have been a longstanding challenge in dynamic symbolic execution and are addressed in the literature in different ways.

In Hybrid Fuzzing, a fuzzer (often coverage-guided) works in tandem with symbolic execution. The low-overhead fuzzer can quickly and efficiently explore easily reachable paths in the program. At the same time, the more heavy-weight symbolic execution can discover paths that random mutation would be unlikely to find due to complex constraints.

Pak et al. [26] show how to use symbolic execution to find a broad set of paths that the fuzzer can explore. Driller [33] proposes using symbolic execution to provide new inputs to the fuzzer when it is unable to make progress. QSYM [38] concentrates on the performance of symbolic execution and demonstrates hybrid fuzzing at-scale by both eliminating an intermediate representation step and reducing the cost of constraint solving—by ignoring difficult constraints. Intriguer [13] shows that existing concolic executors for hybrid fuzzing regularly suffer from unnecessary complex constraints due to input structure invariants and hard-to-solve constraints, resulting in missed bugs. They propose Field-Level Constraint solving to encode input-structure-induced constraints separately and solve them without invoking a constraint solver. Pangolin [19] highlighted that existing hybrid fuzzing systems re-execute inputs from scratch instead of recycling states between similar inputs. They propose Incremental Hybrid Fuzzing with polyhedral path abstraction, which employs an abstract path representation to reuse states among different symbolic execution runs.

Another popular approach to avoid path explosion is the application of **static analyses to guide symbolic execution**. An over-approximation of the interesting paths initially steers symbolic execution, and it is later refined by using the path constraints to validate specific properties.

One of the first approaches to adopt this in a lightweight manner is Directed Symbolic Execution [24]. The authors propose to use the control-flow graph to determine the shortest path to the target point, and then follow it symbolically to reach the target efficiently.

Parvez et al. [27] describe a best-first-search strategy to reach a target line in the program and combine it with static analyses to prune paths that cannot reach the target.

Chopper [34] introduces the concept of chopped symbolic execution. Uninteresting parts of the program are skipped by human annotation, and once a bug is found, the pieces are stitched back together, allowing one to find bugs that are guarded by code that causes state explosions. The authors also propose a set of static analyses to determine which code segments to avoid.

Domain-specific static analyses can also successfully produce paths that are verifiable by symbolic execution (e.g., static analyses that detect multi-reads allow Xu et al. [37] to verify double-fetch bugs across multiple functions in the Linux Kernel).

DSE can succeed on its own when it is constrained to a partial set of functionalities or run at a granularity that allows the system to achieve its scope before the state explosion overwhelms it.

An early general approach to constraining symbolic execution at a function level is Under-Constrained Symbolic Execution [29]. The authors propose to analyze functions individually by allowing for lazy resolution of data structures and pointers. This allows the analysis to detect certain types of bugs, even when the functions expect highly structured arguments.

Selective Symbolic Execution [12] presents a way to seamlessly switch between executing a program symbolically and executing it concretely. This enables symbolic analysis of large and complex programs with non-local handling of symbolic data.

Furthermore, domain-specific approaches can successfully reduce the search space enough for the symbolic execution to produce results (e.g., SYMTCP [35] can synthesize packet sequences that cause network stack operations by selectively executing only the TCP stack code in the Linux Kernel).

Path prioritization has been the focus of a vast body of literature. Since enumerating all the paths in a program can become prohibitively expensive, various techniques exist to guide the exploration toward more promising paths. Common strategies are *DFS*, *BFS*, and *Random Exploration.* These strategies are generic and unsophisticated, but can prove very effective and can help tailor the search to the underlying system (e.g., limiting the memory usage, maximizing the variety of explored paths).

Other works propose heuristics to maximize code coverage [9, 21]. The *Coverage Optimize* search from [9] considers metrics such as the distance from the nearest uncovered instruction to rank states. Similarly, Subpath-Guided search [21] prioritizes subpaths in the control-flow graph that have been visited fewer times. Avgerinos et al. [2] discuss a bug-driven search heuristic that prioritizes loop iterations to find memory corruption errors. Other works [36, 40] use different techniques to select paths that are more suitable to meet a particular property. Zhang et al. [40] use Finite State Machines to prioritize paths that are likely to satisfy a pre-defined regular property. Xie et al. [36] use fitness functions to infer the likelihood of reaching a specific branch.

Our approach has two distinct advantages over existing techniques. First, predicting the likelihood of a path exercising a bug in the program, rather than using a proxy metric like code coverage, implies that improvements to the prediction translate directly to improvements in the number of bugs found. Second, rather than relying on a manually selected set of metrics, SyML automatically learns which features most accurately predict vulnerable code, and can do so on a per-program basis.

Finally, the concept of path prioritization has recently extended to hybrid fuzzing. In DigFuzz [41], the authors derived a probabilistic model to predict how likely the fuzzer is to exercise a path and then prioritized predicted hard paths for symbolic execution. Chen et al. [11] use source code analysis and UBsan labels to identify seeds that traverse more fragile paths in the program. Similarly, Meuzz [10] uses UBsan labels and machine learning on source code to reason per-program and derive a suitable customized strategy for seed prioritization.

8 CONCLUSION

This paper proposes SyML, a novel prioritization technique that leverages supervised learning algorithms to guide symbolic execution and reach vulnerable program states. SyML can effectively mitigate the path explosion problem and offers significant improvements when compared to the state-of-the-art in path prioritization. We evaluate our technique on the CGC dataset and compare its performance with the current state-of-the-art in path prioritization. Our experiments show that SyML outperforms previous work by finding both more and different vulnerabilities. Additionally, we evaluate SyML on 3 real-world Linux CVEs, showing that the knowledge learned from the CGC dataset can be effectively transferred to unseen Linux binaries.

Still, path explosion remains one of the toughest problems in symbolic execution, and our work shows that there is ample space for improvement in path prioritization techniques. While most approaches suffer from a limited understanding of the program logic, search strategies able to better reason on control flow choices are on the rise, and new, different, and complementary solutions will pave the road for more advanced techniques in automatic vulnerability analysis.

ACKNOWLEDGMENTS

We would like to thank our reviewers for their valuable comments and inputs to improve our paper. This material is based upon work supported by NSF under Award No. CNS-1704253. Research was also sponsored by DARPA under agreements number HR001118C0060 and FA8750-19-C-0003. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the U.S. Government, or the other sponsors.

REFERENCES

[1] 2016. "Your tool works better than mine? Prove it". https://blog.trailofbits.com/2016/08/01/your-tool-works-better-than-mine-prove-it/

[2] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (2014), 74–84.

[3] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing symbolic execution with veritesting. *Proceedings of the 36th International Conference on Software Engineering* (2014), 1083–1094.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[5] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. *USENIX Annual Technical Conference, FREENIX Track* 41 (2005), 46.

[6] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[7] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: Attacking path explosion in constraint-based test generation. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), 351–366.

[8] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *OSDI* 8 (2008), 209–224.

[10] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. 2020. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. *arXiv preprint arXiv:2002.08568* (2020).

[11] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Long Lu, et al. 2019. SAVIOR: Towards Bug-Driven Hybrid Testing. *arXiv preprint arXiv:1906.07327* (2019).

[12] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. 2009. Selective Symbolic Execution. *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)* CONF (2009).

[13] Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. 2019. Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing. *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2019). https://doi.org/10.1145/3319535.3354249

[14] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. *2016 IEEE Symposium on Security and Privacy (SP)* (2016), 110–121.

[15] Pedro M Domingos. 2012. A few useful things to know about machine learning. *Commun. ACM* 55, 10 (2012), 78–87.

[16] Dawson Engler and Daniel Dunbar. 2007. Under-constrained execution: making automatic code destruction easy and scalable. *Proceedings of the 2007 international symposium on Software testing and analysis* (2007), 1–4.

[17] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated Whitebox Fuzz Testing. *NDSS* 8 (2008), 151–166.

[18] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. 2020. SYMBION: Interleaving Symbolic with Concrete Execution. *Proceedings of the IEEE Conference on Communications and Network Security (CNS)* (2020).

[19] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. PANGOLIN: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2020).

[20] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. *ACM Sigplan Notices* 47, 6 (2012), 193–204.

[21] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering symbolic execution to less traveled paths. *ACM SigPlan Notices* 48, 10 (2013), 19–32.

[22] Martin C Libicki, Lillian Ablon, and Tim Webb. 2015. *The defender's dilemma: Charting a course toward cybersecurity.* Rand Corporation.

[23] Yu Liu, Xu Zhou, and Wei-Wei Gong. 2017. A Survey of Search Strategies in the Dynamic Symbolic Execution. *ITM Web of Conferences* 12 (2017), 03025.

[24] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings* 6887 (2011), 95–111. https://doi.org/10.1007/978-3-642-23702-7_11

[25] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017), 377–389.

[26] Brian S Pak. 2012. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. *School of Computer Science Carnegie Mellon University* (2012).

[27] Riyad Parvez, Paul A. S. Ward, and Vijay Ganesh. 2016. Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries. *Proceedings of the Annual International Conference on Computer Science and Software Engineering (CASCON)* (2016). http://dl.acm.org/citation.cfm?id=3049889

[28] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12, Oct (2011), 2825–2830.

[29] David A. Ramos and Dawson R. Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. *Proceedings of the USENIX Security Symposium* (2015). https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

[30] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended symbolic execution on binary programs. *Proceedings of the eighteenth international symposium on Software testing and analysis* (2009), 225–236.

[31] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. *2016 IEEE Symposium on Security and Privacy (SP)* (2016), 138–157.

[32] Jia Song and Jim Alves-Foss. 2015. The DARPA cyber grand challenge: A competitor's perspective. *IEEE Security & Privacy* 13, 6 (2015), 72–76.

[33] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. *NDSS* 16 (2016), 1–16.

[34] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. *Proceedings of the International Conference on Software Engineering, (ICSE)* (2018). https://doi.org/10.1145/3180155.3180251

[35] Zhongjie Wang, Shitong Zhu, Yue Cao, Zhiyun Qian, Cheng-Yu Song, S. V. Krishnamurthy, Kevin S. Chan, and Tracy D. Braun. 2020. SymTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery. (2020).

[36] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. *2009 IEEE/IFIP International Conference on Dependable Systems & Networks* (2009), 359–368.

[37] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2018). https://doi.org/10.1109/SP.2018.00017

[38] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. *Proceedings of the USENIX Security Symposium* (2018).

[39] Michal Zalewski. 2014. "American Fuzzy Lop". https://lcamtuf.coredump.cx/afl/

[40] Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. 2015. Regular property guided dynamic symbolic execution. *Proceedings of the IEEE International Conference on Software Engineering (ICSE)* (2015).

[41] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019* (2019).