StatSym: Vulnerable Path Discovery through Statistics-Guided Symbolic Execution

Fan Yao, Yongbo Li, Yurong Chen, Hongfa Xue, Tian Lan and Guru Venkataramani
Department of Electrical and Computer Engineering
The George Washington University
Email: {albertyao, lib, gabrielchen, hongfaxue, tlan, guruv}@gwu.edu

***Abstract*—Identifying vulnerabilities in software systems is crucial to minimizing the damages that result from malicious exploits and software failures. This often requires proper identification of vulnerable execution paths that contain program vulnerabilities or bugs. However, with rapid rise in software complexity, it has become notoriously difficult to identify such vulnerable paths through exhaustively searching the entire program execution space. In this paper, we propose *StatSym*, a novel, automated Statistics-Guided Symbolic Execution framework that integrates the swiftness of statistical inference and the rigorousness of symbolic execution techniques to achieve precision, agility and scalability in vulnerable program path discovery. Our solution first leverages statistical analysis of program runtime information to construct predicates that are indicative of potential vulnerability in programs. These statistically identified paths, along with the associated predicates, effectively drive a symbolic execution engine to verify the presence of vulnerable paths and reduce their time to solution. We evaluate *StatSym* on four real-world applications including *polymorph*, *CTree*, *Grep* and *thttpd* that come from diverse domains. Results show that *StatSym* is able to assist the symbolic executor, KLEE, in identifying the vulnerable paths for all of the four cases, whereas pure symbolic execution fails in three out of four applications due to memory space overrun.**

I. Introduction

Securing software systems has become very challenging due to the growing software complexity. Prior studies have shown that there are about 5 to 20 bugs per 1,000 lines of software code [1]. Exploitation of such bugs and program vulnerabilities has been a major threat to computer security and user data safety. Solution approaches, that effectively address such threats, need to be aware of program paths containing vulnerabilities.

Note that various parts of the software code are likely developed by different programmers over time, and as such, identifying vulnerable program paths can be difficult. Additionally, attackers have become very adept at exploiting vulnerabilities as soon as they are exposed to them [2]. These considerations call for an efficient and scalable approach to analyzing vulnerable program paths in a swift and rigorous manner.

Prior work on program vulnerable path analysis can be broadly classified into three categories [3]: 1. *Record and Replay* systems that perform an exact reproduction of the execution path leading to a certain vulnerability [4], [5], [6], [7]; 2. Techniques that use random testing (e.g., fuzzing and *statistical* methods [8], [9], [10], [11], [12]) relying on sampling of runtime program states and probabilistic representations of knowledge base; 3. Techniques that utilize formal methods (e.g. symbolic execution and dynamic tainting analysis [13], [14], [15], [16], [17], [18]) relying on program models, semantics and logical structures to construct knowledge base.

Unfortunately, all of these prior techniques have some fundamental limitations. As instances, 1. *Record and Replay* requires capturing all program inputs and execution states and incurs high performance overhead (for logging). Plus, there is a considerable deployment cost (to monitor various system components from different vendors), and imprecision due to data unavailability in certain cases (where users do not wish to reveal personal details) [19], [20], [21], [22]. 2. *Pure statistical methods* rely on probabilistic inference and often fail to guarantee complete accuracy. Any conclusions derived from sampling the runtime program states can offer only limited visibility, and are prone to false alarms [9]. As a result, considerable human effort is still required to verify the results from statistical analysis. 3. *Formal methods* require exhaustive analysis along all paths in the application code, that can be prohibitively expensive in terms of time and resources. As such, strict symbolic execution methods can be less effective in analyzing software at-scale.

In this paper, we propose $StatSym$, a framework for vulnerable path identification through *Statistics-Guided Symbolic Execution*. $StatSym$'s efficiency stems from a novel integration of statistical analysis and symbolic execution techniques. First, it leverages statistical analysis to construct program predicates (e.g. conditions and assertions of program states that are indicative of program vulnerability), and then employs a path construction algorithm to select (and rank) the most likely execution path responsible for vulnerability. The suspicious execution path and its associated predicates provide inference to a symbolic execution module, which will perform statistics-guided path exploration in a pruned search space (where higher priorities are assigned to the candidate paths and their close neighbors) to validate the presence of vulnerable paths.

$StatSym$ harnesses the advantages of both statistical analysis and symbolic execution techniques to perform rigorous path detection while maintaining scalability and swiftness. To the best of our knowledge, this is the first work that integrates statistical analysis and symbolic execution to expedite the vulnerable path identification. We evaluate $StatSym$ using real-world software code and demonstrate the usefulness of

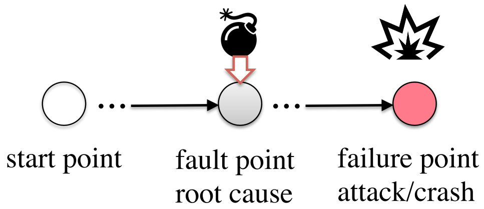


Fig. 1: Conceptual model of software failure. System failure may denote security exploitation or program crash.

our approach in terms of both efficiency and accuracy in identifying the actual vulnerable program paths.

Identifying the *execution paths and constraint sets that lead to vulnerable program states* has several useful applications such as (i) automatically patching the buggy code through hardening of vulnerable paths, (ii) preemptive input filtering/checking to prevent the program from reaching vulnerable states that lead to malicious exploitation or termination, and (iii) providing useful insights to the programmers during debugging phases of code development. In real-world software systems such as web servers (like thttpd [23]), vulnerable inputs can lead to program crashes or undesired information outflows. Identifying the conditions on program variables and/or execution paths that can lead to such vulnerable system states, is helpful to track and mitigate the malicious effects of such exploits.

In summary, the contributions of this paper are:

1) We design and implement *StatSym*, a novel *automated* framework for vulnerable path identification by integrating statistical analysis and symbolic execution, ultimately reducing the *time to solution*.

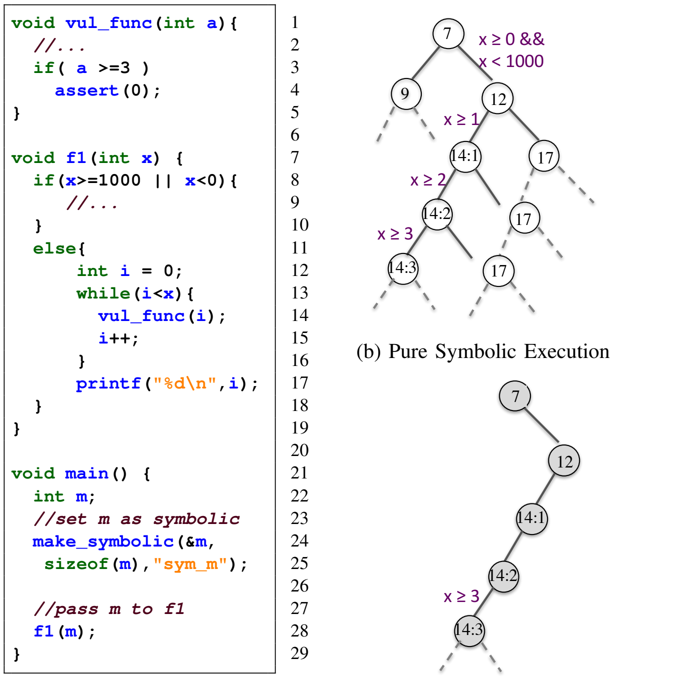
2) *StatSym* employs a new vulnerable path construction algorithm that leverages statistical inference to construct vulnerable paths with partial execution logs (typical of most real-world situations). The suspicious execution paths identified by statistical analysis guide symbolic execution to search vulnerable paths efficiently.

3) We evaluate the proposed *StatSym* using real-world applications from diverse domains: *polymorph* [24], *CTree*, *Grep* [25] and *thttpd* [23]. From our experiments, we observe that *StatSym* is able to identify the vulnerable paths in all the four programs. In *polymorph*, *StatSym* speeds up the vulnerable path discovery by about 15× over pure symbolic executor, KLEE [13].

4) We perform sensitivity studies on how different program logging and sampling impact the performance of *StatSym*. Our results show that *StatSym* is able to identify vulnerable paths and achieve significant speedup even at relatively low sampling rate of 20%.

Program	SLOC	Ext. Call	Inter. Call	G.V.	Params.
polymorph	506	29	16	36	253
CTree	3011	50	52	188	1568
Grep	6660	143	532	718	15760
thttpd	7939	114	145	545	7420

TABLE I: Program source statistics: Source Lines of Code (SLOC), External Calls (Ext. Call), Internal user-level calls (Inter. Call), Global Variables (G.V.) and Function parameters (Params.).



```
void vul_func(int a){
  //...
  if( a >=3 )
    assert(0);
}

void f1(int x) {
  if(x>=1000 || x<0){
     //...
  }
  else{
      int i = 0;
      while(i<x){
        vul_func(i);
        i++;
      }
      printf("%d\n",i);
  }
}

void main() {
  int m;
  //set m as symbolic
  make_symbolic(&m,
   sizeof(m),"sym_m");

  //pass m to f1
  f1(m);
}
```

(a) Sample source code

(b) Pure Symbolic Execution

(c) Statistics-Guided Symbolic Execution

Fig. 2: Illustration showing pure symbolic execution's search space being reduced greatly through statistics-guided symbolic execution in *StatSym*.

II. Motivation

Consider the system failure model shown in Figure 1. During vulnerable path analysis, there are two important points in the execution path: the fault point and the failure point [26]. Fault point is the root cause of failure (*e.g. location where a* ***strcpy*** *is performed without checking the length of target string buffer*). Failure point is the point of undesired program outcome (e.g., program crash or a manifested malicious attack such as sensitive information exfiltration). In actual software systems, these two points can be far away from each other. For instance, in the case of return-oriented programming [27], consider a function $\mathcal{F}$ with a string $\mathcal{P}$. The fault point can be a statement in function $\mathcal{F}$ that fills the memory pointed by $\mathcal{P}$ using an adversary's malicious inputs. The payload injected by the adversary would only be executed when returning from function $\mathcal{F}$ (e.g., the failure point).

In fact, there are several documented evidences of bugs in real-world applications that have led to security attacks. In this paper, we consider four buggy applications: *polymorph*, a utility tool for file name conversion; *CTree*, a GNU tool for displaying file system hierarchy; *Grep*, a command-line utility for plain-text search and *thttpd*, an open source web server.

Thttpd, the server-class application with several thousand lines of code, has numerous documented bugs. Reports from CVE [28] show that several security vulnerabilities exist in the release *2.25b* that allow remote code execution. Examples include string replacement buffer overflow vulnerability in a function named *defang* [28]. *CTree* and *Grep* from the STONESOUP project [25], with thousands of lines of code (not including external libraries) and several thousands of variables and function parameters, have various types of vulnerabilities including memory corruption and integer overflows. *polymorph* from Bugbench [24], contains a stack buffer overrun vulnerability.

Identifying vulnerable paths in applications using pure symbolic execution may be impossible due to *path explosion* problem. As the scale of the programs become large, the number of possible exploration paths can quickly grow exponentially depending on the number of intervening branches, loops and recursive functions embedded in the program source code. This would quickly exhaust both computation and storage resources in computer systems, limiting its capability to identify vulnerable program paths. Table I shows statistics from various applications in terms of lines of code, number of function calls, variables and function parameters.

In contrast to symbolic execution, statistical testing methods rely on logging of program states for analysis. Global variables, function parameters, return values, and occasionally, certain local variables are components of such program states. To keep performance overheads acceptable, existing statistical methods generally advocate for partial logging through sampling [9]. However, an adverse side-effect of partial/incomplete logging is that, it becomes extremely hard for statistical methods alone to accurately identify the predicates and program paths that lead to vulnerable execution, resulting in one or both of the following undesirable outcomes: 1. missed vulnerable paths (false negatives) due to inadequate statistical profile data, or 2. weak or wrong predicates due to imperfect statistical data. Consequently, manual reviews and debugging of statistical output are still required for further verification.

In summary, we note that, a good solution approach to program vulnerability analysis requires two qualities: *efficiency* or *effectiveness*. *Efficiency* is the ability for a solution to make swift inference, while *Effectiveness* defines the rigorousness and accuracy of vulnerable path identification. We note that statistical methods are efficient (performance-wise) but are less effective in locating the exact vulnerable program paths. On the contrary, pure symbolic execution methods are more effective with their rigorous analysis and rule-based models, but the performance overheads are prohibitively high. This motivates our design of $StatSym$ framework, which pushes the envelope for vulnerable path identification through a novel integration of the symbolic execution and statistical approaches. More precisely, the statistical analysis module of $StatSym$ constructs predicates and candidate vulnerable paths that are indicative of program vulnerability and guides the symbolic execution module through pruning its search space. In this way, symbolic executors can assign higher search priority to suspicious candidate paths (and their close neighbors) until the vulnerable paths are assertively identified and the associated path constraints are generated.

Consider a simple C program in Figure 2a consisting of some conditional operations and loop iterations for an integer input x. There is an assertion statement in $vul\_func()$ associated with the range of the argument a (a $>=$ 3), which is guarded by x in the *while* loop in line 13. To search for a vulnerable path, pure symbolic execution tools would typically set x as symbolic and *fork* a *brand new* state after each loop iteration. The corresponding path exploration space is shown in Figure 2b as a tree structure. As we can see, a symbolic executor has to explore both sets of execution paths for the *if* statement in line 8. This exploration has to be repeated for every iteration of the loop.

The statistical approach in $StatSym$ is able to identify the most likely vulnerable path by constructing predicates and conditions for the target vulnerability. In this example, it first selects a candidate path covering 7→12→14 using program execution history (both faulty and non-faulty). This helps trim off[1] the unnecessary subtree of states on the left side with root node 9. Then, $StatSym$ infers that the range of values for variable x which is highly correlated with the vulnerability. As x can directly guide the number of iterations to be considered by the symbolic executor in line 13, a large number of subtrees corresponding to the paths (that are unlikely to be responsible for the vulnerability) are trimmed. The pruned subtree in Figure 2c illustrates the reduced search space with $StatSym$ when the candidate path is 7→12→14 with a predicate of ($x \geq 3$). With statistical guidance, we see that the actual states/paths that need to be explored by $StatSym$ are significantly reduced when compared to that of pure symbolic execution.

III. Threat Model and Assumptions

A. Threat Model

In this work, we consider program vulnerabilities including software bugs and defects that can manifest by exploiting program control flows or by manipulating program inputs or states. When these vulnerabilities are exploited, their effects will manifest and lead to program crashes or security breaches. Identifying vulnerable paths leading to the program failure is crucial for hardening these software systems, e.g., by filtering faulty inputs, path-sensitive analysis [29], [30] and hardening code path [2], [31]. In case of more sophisticated attacks, such as silent corruption of memory values, we note that our proposed mechanism can still be just as effective, provided that the point of attack (data corruption) can be identified through periodically imaging the memory. Note that vulnerable path discovery has application in software debugging as well.

[1] The non-selected paths (that are trimmed off) receive lower priority in symbolic execution. So, in the (unlikely) worst case when erroneous statistical inference is made, the performance of $StatSym$ is equivalent to pure symbolic execution.

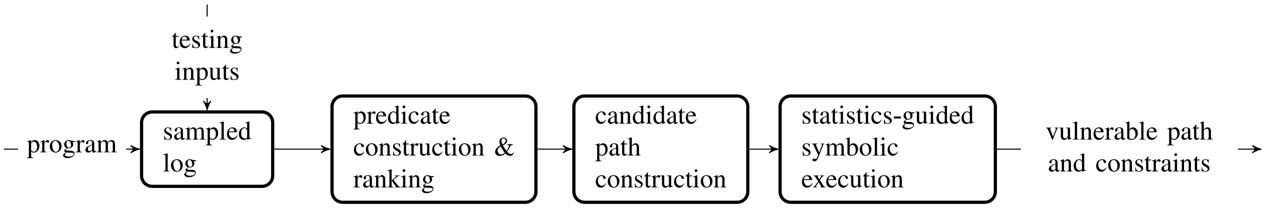


Fig. 3: Design Overview of Vulnerability Discovery Framework based on Statistics-Guided Symbolic Execution.

B. The Use of Partial Logging

We assume that only partial logging is available at runtime, which provides incomplete inputs for statistical analysis in $StatSym$. This assumption is based on findings in numerous prior works that demonstrate full logging to be impractical in real-world applications [8], [9], [10], [11]. In order to reduce the performance overhead for runtime logging, prior works often take advantage of statistical sampling to log only partial information on the fly. In this paper, we consider logging only at function entry and exit points, and our logging targets include program global variables, function parameters and return values. We note that some program variables may contain sensitive user information. Thus, the logging system may define security rules to prevent collecting such information, e.g., logging only the length of string objects and hashing function names. To preserve user privacy, Yuan et al. [21] show how users can be provided with the option to inspect the runtime logs and eliminate any sensitive information.

C. Existence of Multiple Vulnerabilities and Paths

Multiple vulnerabilities may exist in a single program, especially as program size increases. While this paper focuses on identification of single vulnerable path, the proposed $StatSym$ framework can be easily extended to programs with multiple vulnerabilities and/or multiple vulnerable paths. Prior works have studied bug isolation techniques [9], and leverage machine learning and clustering techniques [11] to separate log files pertaining to different bugs to generate statistical inference for each individual bug. By taking advantage of these techniques, we can isolate different vulnerabilities and use $StatSym$ to identify (and eliminate) vulnerable paths one-by-one through an iterative process until all vulnerabilities and paths are identified.

IV. Approach Overview

$StatSym$ makes use of statistical analysis to 1. infer candidate paths that are most likely related to the vulnerability and 2. construct and rank predicates (i.e., conditions associated with program states) based on their relevance to program failures or security breaches. The results effectively guide path-driven symbolic execution to concentrate on more likely candidate paths with higher priority, thus improving the efficiency of vulnerable path identification.

$StatSym$ has the following components (Figure 3):

- A runtime sampling and logging component that collects runtime program information including *global variables*, *function parameters* and *return values* at function entry and exit points. To minimize the effects of logging and the associated performance overheads, we adopt a probabilistic sampling method.
- A statistical inference component which analyzes the runtime program information, constructs predicates (conditions and certain assertions about program states or variables), and ranks them based on relevance to the program failure point. A higher confidence score indicates a higher likelihood of a predicate being indicative of the target vulnerability.
- A candidate path construction component that generates candidate vulnerable paths that will receive higher priority for symbolic execution based on the confidence scores. Each node in the path represents a *function entry* or *function exit* point (instrumentation location).
- A statistics-guided symbolic execution component that performs program state exploration based on vulnerable paths. Once the actual vulnerable path is found, the symbolic execution procedure would output the complete execution path (and path constraints) that leads to the program failure point.

V. **StatSym** Design

This section presents our $StatSym$ framework and algorithm design, including (i) predicate construction and ranking, (ii) candidate path identification and (iii) statistics-guided symbolic execution.

A program can be represented at different granularities. In a Control Flow Graph (CFG), each node represents a basic block (a straight-line sequence of code without any branches except at the entry and exit points), and each directed edge denotes a possible control transfer (e.g., conditional branches). At a coarser granularity, a program can also be represented by a Call Graph (CG), where each node corresponds to a function or procedure, and an edge represents the call relations between functions or procedures. Note that CG represents a subset of CFG information since function calls and returns are also part of control transfer instructions. In this paper, we use a graph representation $\mathcal{G}$ to denote a program that primarily considers a CG due to the limitations stemming from the tools used in our studies (See Section VI for further details). We note that our framework can be easily extended to include finer granularity CFG nodes as well if we have the capability to instrument the program at the basic block block level, and monitor variables inside these basic blocks. Without loss of generality, we will refer to the granularity of observation as program blocks.

The graph $\mathcal{G}$ consists of a node set $\mathcal{N}$ and an edge set $\mathcal{E}$. We consider program blocks, each consisting of program code between two instrument points and represented by a node $n_i \in \mathcal{N}$ in $\mathcal{G}$. The edges in $\mathcal{E}$ are directed and denoted by (n_i, n_j) where $n_i \in \mathcal{N}$ and $n_j \in \mathcal{N}$ is the head and tail of the edge respectively. A state comprises a set of variables and program control transfers that are made visible by source-level or runtime program instrumentation.

A. Predicate Construction and Ranking

We identify a list of instrumentation locations, program variables and statements for predicate construction. Such runs are provided to $StatSym$ by the users, which are then used by the statistical sampling module. Our evaluation emulates average user behavior by providing a series of random inputs to the applications, generating a sufficiently large number of sample runs, and randomly sampling them to assemble a set of correct executions and faulty executions. Also, to differentiate program execution states, the same variable instrumented at different locations is considered separately. For example, a global variable that appears in two different physical locations in the program are considered separately from each other.

Predicate Construction: We analyze each variable's statistics (e.g., distributions) in correct and faulty executions, and construct the predicate based on the divergence between the variable's statistics in these two cases, e.g., offering highest degree of distinction between them. More precisely, if variable a at a certain location has a set $\mathcal{C}$ of values in correct executions and a set $\mathcal{F}$ of values in faulty executions, formally, we construct a predicate $x = \{a \in \mathcal{P}\}$ for variable a to optimally separate the instances of a within correct executions and faulty executions, by minimizing the quantification error:

$$E = |\mathcal{P} \cap \mathcal{C}| + |\mathcal{P}^c \cap \mathcal{F}|, \tag{1}$$

where $\mathcal{P}^c$ is the compliment of $\mathcal{P}$. Intuitively, the predicate x provides an optimal separation between correct executions and faulty executions using the distributions of variable a. For example, if an integer variable a in faulty executions is always larger than that of correct executions, we can find a threshold σ that separates the two sets and construct a predicate as $x = \{a \geq \sigma\}$.

Predicate Confidence Score and Ranking: We derive a confidence score metric to measure the capability of a predicate in indicating vulnerabilities. Predicate x receives a higher score if it can better distinguish faulty from correct executions. Let $\mathbb{P}(x|\mathcal{C})$ be the probability (i.e., implied frequency from the log files) of predicate x being true in correct executions, and $\mathbb{P}(x|\mathcal{F})$ be the probability in faulty executions. The absolute difference between these two probabilities is assigned as the temporary score for the predicate of that specific instrumentation location and variable:

$$s = |\mathbb{P}(x|\mathcal{C}) - \mathbb{P}(x|\mathcal{F})|. \tag{2}$$

The larger a predicate score is, the higher likelihood vulnerability is associated with the variable and instrumented location involved in predicate x.

B. Candidate Path Identification

We propose a path identification algorithm to statistically extract program execution paths, traversing locations of highly ranked predicates and leading to the vulnerability manifestation (failure) point. First, due to possibly incomplete program profile from probabilistic logging, we need to construct the transitions between different instrumented locations from faulty executions, which is modeled as an *association rule mining* problem [32], [31]. Let $o(e_i)$ and $o(e_j)$ be the number of occurrences for instrumented locations e_i and e_j in all the log files, we calculate the frequency that e_j occurs after e_i, denoted by $o(e_i \rightarrow e_j)$. This enables us to calculate the confidence μ of transition $e_i \rightarrow e_j$:

$$\mu(e_i, e_j) \quad = \tfrac{o(e_i \rightarrow e_j)}{o(e_i)} \tag{3}$$

Through identifying all transitions with statistically significant confidence scores, we are able to construct a transition graph for each tested program. We propose a heuristic-based path identification algorithm to extract candidate paths traversing the instrumented locations of high confidence-score predicates. The path identification involves three steps: 1. Find acyclic paths starting from nodes that have no incoming arcs (e.g. representing possible program entry points) to the failure point. The score of each node on the path is represented by the predicate with the highest score associated with the instrumented location. From all the acyclic paths, we select the one with the largest average confidence score. Such a path is referred to as *skeleton* in our algorithm. 2. There could be predicates with high confidence scores that are not included in the skeleton. To account for these predicates, we use a greedy algorithm to identify the path segments that branch out from the skeleton and traverse these high confidence-score predicates. The path segments linking the skeleton and high confidence-score predicates are called *detours*. 3. A candidate path is constructed by combining the skeleton and detours, allowing a search to visit high confidence-score predicates on detours, while moving along the skeleton toward vulnerability manifestation point.

C. Statistics-Guided Symbolic Execution

A candidate path includes a sequence of instrumented locations and the associated predicates. An example of the candidate path consisting of multiple nodes is shown in Figure 4. The dotted-gray circle is the starting point of the path (e.g. *main()* function). Each subsequent node (e.g., green circles marked by n_i) represents an instrumented location in the program, associated with a predicate p_i. The vertical-dashed red circle denotes the failure point.

Note that the candidate paths generated from statistical analysis might not be a viable path during actual program execution due to incomplete statistical information. To guarantee the accuracy of discovered vulnerability paths, symbolic executors are used to verify statistically-identified candidate paths. As shown in Figure 4, $StatSym$ utilizes two mechanisms to guide symbolic execution: *inter-function* and *intra-function* search. Inter-function search is guided by the nodes

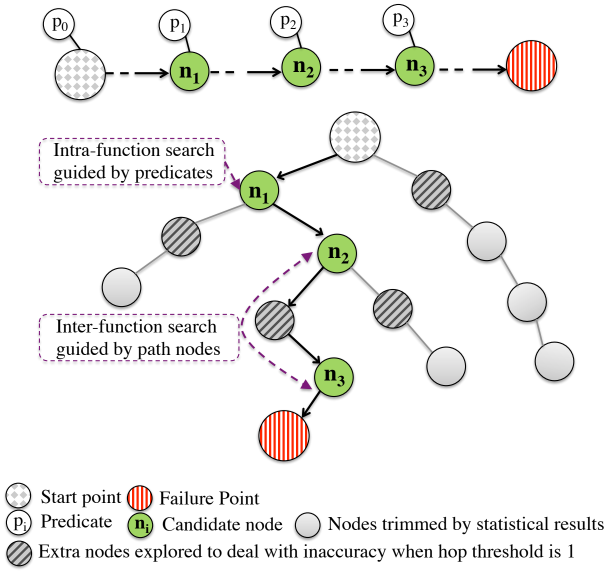

p0
p1
p2
p3
n1
n2
n3
Intra-function search guided by predicates
Inter-function search guided by path nodes
Start point
Failure Point
Predicate
Candidate node
Nodes trimmed by statistical results
Extra nodes explored to deal with inaccuracy when hop threshold is 1

Fig. 4: Candidate path through statistical analysis (top) and the symbolic executor's search space (bottom).

in the candidate path. The symbolic executor selects those execution paths that follow a given candidate path and *does not* deviate by more than a *preset number of hops*. In Figure 4, the diagonal-filled circles denote nodes that are within the allowed hop distance. Intra-function search is guided by leveraging predicates. When symbolic execution spawns additional states at branch points, the constraints associated with the predicate at the location combined with the constraint set (including the branch conditions) would be evaluated. If the constraints and predicates are not satisfied, the state associated with the current branch will be explored at a later stage when no other active states are available. This enables $StatSym$ to significantly trim down the search space by eliminating paths that either do not follow a candidate path or that are unlikely to be related to program vulnerability (denoted by the gray circles in Figure 4).

To improve the possibility of finding vulnerable program paths, we allow symbolic executors to search paths that deviate from the candidate path within a threshold of τ hops. For example, if on a candidate path, two locations a and b have no direct links in the program, the statistics-guided symbolic execution module will start to search with a maximum depth equal to τ to reach location b from location a. A higher τ improves the robustness of symbolic executors at the expense of increasing runtime overhead. The example in Figure 4 has threshold τ set to 1. If a feasible solution exists for the constraints involved in an execution path, the symbolic executor validates that the path and confirms its association with vulnerability. The verified path and the associated constraints would be output as the result. The overall algorithm governing the operations of different modules is shown in Figure 5.

VI. Implementation

We implement a prototype of $StatSym$ that has two modules: a statistical analysis module and a symbolic execution module (Figure 6). Statistical analysis module consists of two sub-systems, Predicate Manager and Candidate Path Constructor Symbolic execution module is built on KLEE [13].

```
Algorithm: Statistics-Guided Symbolic Execution
Preprocess log files, count the numbers of runs $n(R)$,
locations $n(L)$ and logged variables $n(V)$
(a) Divide the $n(R)$ of runs into correct executions $\mathcal{C}$
  and incorrect executions $\mathcal{I}$
(b) Transform the logged data
  transform non-numerical variables' characteristics to
  numerical values
(c) Predicate construction
for all logging locations $R$
  for all logging variables $L$
    (c.1) Construct a predicate $x$ based on $\mathcal{C}$ and $\mathcal{I}$
    (c.2) Calculate probabilities for when the predicate is true
        within $\mathcal{C}$ and $\mathcal{I}$, i.e. $p(x,\mathcal{C})$ and $p(x,\mathcal{I})$
    (c.3) Assign $|p(x,\mathcal{C}) - p(x,\mathcal{I})|$ as score
  end for
end for
(d) Rank $|p(x,\mathcal{C}) - p(x,\mathcal{I})|$ for all $R$ and $L$
(e) Symbolic execution
  set up timer for symbolic path exploration
  while (problematic path is not found & timer has not expired)
    (e.1) Join skeleton and detours to get a candidate
        path $\mathcal{P}$ with the largest average predicate score
    (e.2) Do Symbolic execution using candidate path ($\mathcal{P}$)
    (e.3) if bug is triggered in (e.2)
          Output the complete path of (e.2) and the associated
          predicates
        end if
  end while
```

Fig. 5: $StatSym$ Algorithm

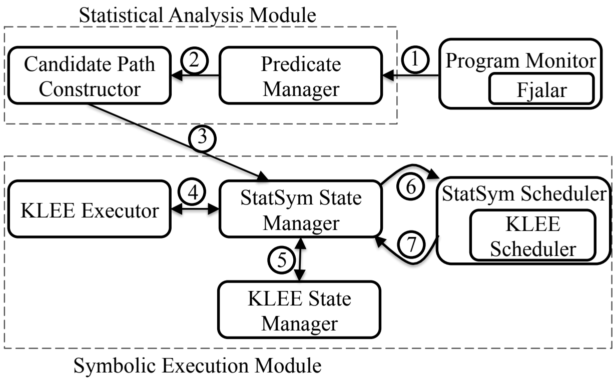


Fig. 6: $StatSym$ System Framework

A. Program Monitor

We implement our program monitor using Valgrind [33]. We build a custom instrumentation tool by modifying Fjalar [34] which is a plug-in on top of Valgrind. Fjalar is able to dynamically instrument on unmodified C/C++ programs and provides rich source code-level semantic information such as variable names, variable types and function names. By overriding the instrumentation behaviors at function entry and exit points, our custom tool is able to log variable names and values based on their type. Leveraging the built-in support from Fjalar, our program monitor is able to perform runtime logging at tunable sampling rates, allowing for partial logging.

B. Statistical Analysis Module

The Statistical analysis module includes two components: *Predicate Manager* and *Candidate Path Constructor* (implemented with around 3K Lines of Code using Python). The *Predicate Manager* reads multiple runtime logs generated by the program monitor ①, and then constructs predicates and ranks them based on confidence scores that measure the statistical difference of a variable (and its values) between correct executions and faulty executions. Note that a higher ranked predicate indicates a closer correlation with program vulnerability.

The *Candidate Path Constructor* first identifies the transition links between different locations (using Equation (3)) and constructs a dynamic control flow graph connected via multiple function call entries and exit points. The *Candidate Path Constructor* will utilize the ranked predicates from *Predicate Manager* ② to extract a *skeleton* and the associated *detours*. A skeleton is obtained by choosing the path with highest average predicate score when breadth first search is performed starting from the program entry point to the failure point.

The candidate vulnerable paths are constructed by joining the skeleton and detours based on predicate scores to improve vulnerable path discovery. As discussed in Section V-B, each detour can start from a certain node on the skeleton that finally returns back to the skeleton. Depending on the indices of the starting and ending nodes of a detour in the skeleton, they can be categorized into three types: (i) detours whose starting index is smaller than the ending index on the skeleton, (ii) detours with the starting index larger than the ending index, and (iii) detours with the starting and ending nodes at the same index on the skeleton. The latter two detour types will introduce cycles to the candidate path, and the first type of detours may replace certain segments of the original skeleton. In our current implementation, we apply different heuristics that are aware of these detour types. For example, for each unique location on the skeleton, if there exist multiple detours of the same type, we calculate the average predicate scores for all such detours, and select the one with largest confidence score.

C. Statistics-Guided Symbolic Execution

Symbolic Execution in KLEE: Each path that is to be explored by KLEE at runtime, is represented by a unique state and identified using the *traversed branch decisions*, *current program counter* and *stack frames*. The LLVM bitcode interpreter or *Executor* uses a loop to iterate over instructions (*stepInstruction()*) and executes them (*executeInstruction()*) symbolically if possible. Upon reaching a branch instruction (e.g. {*if, else*}, {*switch, case*}), the executor will attempt to fork a state, provided that the constraint corresponding to the branch direction is satisfiable. KLEE implements several state scheduling algorithms including Breadth First Search (BFS), Depth First Search (DFS), Random Path Selection and even a sophisticated coverage-optimized search that uses a heuristic to weigh each state based on the likelihood of covering new source code.

Statistics-Guided Symbolic Execution in $\mathtt{StatSym}$: We modify KLEE to take advantage of the candidate path and predicates information output by statistical analysis module to the State Manager of $StatSym$ ③. The $StatSym$ *State Manager* infers and maintains all of the potential states that are worthy of subsequent exploration. In our implementation, the KLEE Executor retrieves one state a time from the $StatSym$ *State Manager* and executes the next instruction for that state. The $StatSym$ *State Manager* records the progress of symbolic execution along the candidate path for all states by bookmarking the currently executed path nodes, as well as the diverted hops (Section V-C). More importantly, the $StatSym$ *State Manager* will use the candidate path to guide state exploration: *Inter-function* and *Intra-function*. When the current symbolic execution point matches the name of the candidate path node, the constraints indicated by predicates would be added to the current state in $StatSym$ *State Manager*[2], which helps trim down intra-function search space. The symbolic execution states corresponding to the branch instruction outcomes that conflict with the constructed predicates will be suspended. For inter-function guidance at function entry and exit, the *State Manager* will update the diverted hops in the current state, and suspend the states that have exceeded the hop diversion threshold from further consideration. $StatSym$ *State Manager* also coordinates with KLEE's default state manager to maintain necessary information for each state such as constraint set and memory space. The $StatSym$ *State Scheduler* interacts with the *State Manager* to select the next state to be explored, ⑥, ⑦. The $StatSym$ scheduler gives the states that have less diverted hops higher priority. If a candidate vulnerable path is verified by the Symbolic Executor, the complete vulnerable path along with the constraints are output. If not, the next candidate path in the list (if one exists) would be explored, and the above process would be repeated over again. We note that *StatSym* provides the same level of code coverage and depth in the search as compared to KLEE's symbolic execution.

VII. Evaluation

A. Experimental Setup

Benchmark Selection: We performed extensive studies on application benchmarks from multiple sources [24], [25], [35] that include server class, GNU utilities and database system. The vulnerabilities associated with the target programs include buffer overruns, integer handling errors, pointer dereferencing vulnerabilities and data race bugs. To represent diverse domains, we select four real-world programs: polymorph, Grep, CTree and thttpd.

[2] Note that KLEE does not directly support constraining string length. As a workaround, for strings with unknown length, we intentionally allocate long enough memory array. We then constrain the length of the string by controlling the index at which the first '\0' resides.

	Statistical Analysis Module		Symbolic Execution Module
Benchmark	detours	time(sec)	time(sec)
polymorph	0	1.9	180.6
CTree	0	58.4	1.6
thttpd	6	561.2	247
Grep	12	661.4	37.7

TABLE II: Number of detours, and time breakdown when sampling rate is 100%.

	Statistical Analysis Module		Symbolic Execution Module
Benchmark	detours	time(sec)	time(sec)
polymorph	2	1.6	213.0
CTree	1	43.2	2.4
thttpd	7	428.0	1263.0
Grep	31	518.7	44.3

TABLE III: Number of detours, and time breakdown when sampling rate is 30%.

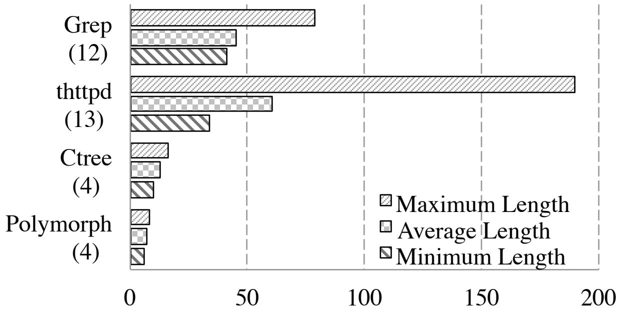


Fig. 7: Statistics of candidate path lengths. The number below each application name represents the total number of paths generated by statistical analysis.

Log Collection and Sampling: Ideally, we need to collect a large set of the user's execution traces, and then randomly sample them to assemble our sets of correct execution logs and faulty execution logs. To emulate this scenario, we run our application with a sufficiently large number of randomly generated input sets that result in correct and faulty executions. We collect logs of the target programs using the Program Monitor described in Section VI-A. Each log file corresponds to a program run and is annotated with a flag indicating whether it is from a correct or faulty execution. Among all of the generated logs, we randomly choose one hundred correct execution logs and one hundred faulty execution logs. In our evaluation, we use partial, incomplete logging at the sample rate of 30% and 100%. For *polymorph* and *CTree*, we also select multiple levels of sampling ranging from 20% to 100% for the sensitivity study in Section VII-C. Note that $StatSym$ is effective even at a relatively low sampling rate of 20%, as will be shown later in Section VII-D.

Symbolic Execution: We use unmodified symbolic execution on KLEE as the baseline for comparison. For $StatSym$, we set the timeout for statistics-guided symbolic execution to be 15 minutes for a given candidate path; we set the symbolic execution timeout length for KLEE to be 8 hours. The default threshold value τ is set to 10 for all applications (See Section V-C for details). For both $StatSym$ and KLEE to perform symbolic execution, proper configuration of the symbolic inputs in each program is required. Valid program inputs typically follow certain format. For example, *CTree* program requires the option of *-n* and *-q*. We note that, without knowledge of the parameter format and option semantics, symbolic executor would spend enormous time iterating over program parameter parsing, which is irrelevant to the program's core functionality. In our experiments, we configure *such semantically reasonable and required* program input options for both $StatSym$ and KLEE to avoid unnecessary (exhaustive) search. Recent works such as [36] highlight the needs to automatically recognize program input constraints for more efficient symbolic execution, and we note that such optimizations are beyond the scope of our work.

Experiment Testbed: All experiments are conducted on our lab server equipped with a 4-core Intel Xeon E5405 server and 12GB DRAM.

B. Evaluation Results

For all of the benchmarks, *StatSym* is able to successfully identify the vulnerable paths correctly corresponding to the respective vulnerabilities. In Table II and Table III, we summarize the time overheads for both statistical analysis and symbolic execution module with two sampling rates, 100% and 30% respectively. We record the time consumed for the statistical analysis and the number of different detours generated. For statistics-guided symbolic execution, we record the time taken to find the vulnerable paths and generate the corresponding test inputs.

From Table II, we can see that the statistical analysis module effectively generates the candidate paths, especially for *polymorph* and *CTree* (with 0 detours). When sampling rate is reduced to 30%, Table III shows that the number of detours increases slightly for these two programs resulting in a higher number of possible candidate paths. Figure 7 shows the distribution of candidate path lengths, including minimum, maximum and average number of nodes (path length) for each target program's candidate paths, which are generated by joining the detours using heuristics explained in Section VI-B. We observed that the first generated candidate path at 30% sampling for both *polymorph* and *CTree* has fewer nodes. For example, the first candidate path output under 100% sampling for *polymorph* includes 10 nodes while the one under 30% sampling only has 6 nodes. Interestingly, even with fewer nodes in the first candidate path, $StatSym$ can successfully find the vulnerability in its first iteration. Also, it is worth noting that the module, that dominates runtime overhead, varies across applications. At 30% sampling, for *polymorph*, the program logs generated are very small that it takes less than 2.0 seconds for statistical method to construct candidate paths. In this case, the statistics-guided symbolic executor dominates the time (180.6s). On the contrary, for *Grep*, we observe that its log size is large (hundreds of MB), the statistical analysis takes much longer than symbolic execution. In Table IV, we present the total amount of time (including statistical analysis and symbolic execution) to identify the vulnerable path for $StatSym$ compared with the time taken for pure symbolic exploration by KLEE. We observe that $StatSym$ always explores significantly less number of paths and spends less time than KLEE to find the vulnerability in the program (on average 85.3% fewer paths). Notably, even with significantly higher number of paths explored, for *CTree* and *Grep*, pure symbolic execution by KLEE fails to find the vulnerability and reports state exploration failure due to *lack of available memory* that stopped the constraint solver from forking more processes. For the smallest program, *polymorph*, under 30% sampling, it takes only 214.6 seconds to find the vulnerability, which is a 15 × speedup compared with pure symbolic execution. Across all four programs, $StatSym$ is able to correctly discover the vulnerable paths and the corresponding inputs with no false positives or false negatives. Moreover, the vulnerable paths for three of the programs are identified using the first candidate path found by statistical analysis module (see Section VII-C for details).

	KLEE w/ $StatSym$		Pure Sym. Exec. w/ KLEE	
Benchmark	#paths	time(sec)	#paths	time(sec)
polymorph	63	214.6	8368	3252.0
CTree	112	45.6	17575	Failed
thttpd	5168	1691.0	17882	Failed
Grep	11462	563.0	38708	Failed

TABLE IV: Number of paths explored and time spent before finding the bug in $StatSym$ with KLEE and pure symbolic execution by KLEE at 30% sampling rate.

Instrumented Locations:

L1: grok_commandLine():leave	L2: convert_fileName():enter
L3: is_fileHidden():leave	L4: does_nameHaveUppers():enter
L5: does_newnameExist():leave	L6: grok_commandLine():enter
L7: convert_fileName():leave	L8: main():enter
L9: does_newnameExist():enter	L10: main():leave
L11: is_fileHidden():enter	L12: does_nameHaveUppers():leave

Instrumented Variables:

GLOBAL: *target*, *wd*, *hidden*, *track*, *clean*, *init_file*, *hidden_file*

FUNCPARAM: *argc*, *original*, *suspect*

Fig. 8: Instrumented locations and variables in *Polymorph*

No.	Predicate	Loc.
P1	len(suspect_FUNCPARAM) >536.5	L9
P2	len(original_FUNCPARAM)>518.0	L2
P3	len(suspect_FUNCPARAM)>535.0	L12
P4	len(suspect_FUNCPARAM)>517.5	L3
P5	len(suspect_FUNCPARAM)>526.0	L2
P6	len(suspect_FUNCPARAM)>497.5	L5
P7	track_GLOBAL<-infinity	L7
P8	wd_GLOBAL<-infinity	L7
P9	track_GLOBAL<-infinity	L10
P10	clean_GLOBAL<-infinity	L10

TABLE V: List of top 10 predicates for the *Polymorph*

C. Case Studies

*1) **Polymorph**:* Polymorph is a file name conversion utility. Users of polymorph can provide either a file name or an entire folder to polymorph for name conversion. Specifically, user provides the option *-f* together with a string as the name of the targeted file for conversion. After parsing the command line parameters, the user provided string name *target* is passed to function *convert_fileName()* as parameter *original*. The vulnerability resides in *convert_fileName()* where each character in *target* is read, transformed (if necessary) and copied to the stack allocated 512-Byte buffer *newName* without boundary check. *convert_fileName()* then performs a series of system calls before it returns. The program will crash if the stack is filled with an adversary's payload.

With 30% sampling rate, there are 12 instrumented locations denoted by L1$\sim$L12 and 10 instrumented variables, shown in Figure 8. Table V shows the top 10 predicates. The numbers in third column of the table denote the instrumented location index in Figure 8. We observe that the first six predicates limit the length of strings that implicitly depend on *target*. When the user-provided file name is longer than 512 bytes, this would trigger the vulnerability in *convert_fileName()*. The last four predicates indicate that in L7 and L10, the constraints are not satisfied to trigger the vulnerability under study. This is because only correct runs could reach *L7* - the return of *convert_fileName()*[3] and *L10* - return of *main()*. Figure 9 shows the ordered candidate paths. The top-most path is provided as the first output candidate path to statistics-guided symbolic execution. The symbolic execution module is able to explore the node *convert_fileName()* which is missing in between L1 and L11 in the real execution path. The predicate set (P1, P5, P3, P6) guides the branching of the major loop iterating over user-provided string *target* in *convert_fileName()* so that the sub-paths corresponding to *strlen(suspect)<536.5* are eventually discarded. The symbolic executor quickly traverses through the path that has excessive write iterations to *newName* and finally the vulnerability is triggered. $StatSym$ then stops exploration at the failure point upon return of *convert_fileName()*. As shown in Table IV, $StatSym$ only explores 63 paths until the final vulnerable path is discovered.

*2) **thttpd**:* *thttpd* is a widely-used web server application. A buffer-overflow bug in version *2.25* [23] is triggered when *defang* function processes input variable 'str' derived from user input. After replacing '<' and '>' in 'str' with '&lt;' and '&gt;', the manipulated string is copied to a destination string variable called 'dfstr' and will potentially cause remote code injection. From Table IV we see that pure symbolic execution failed after exploring almost 18K individual paths. Two main reasons exist for the failure: First, thttpd involves large number

[3] In the case of faulty runs, our monitoring tool will not capture the function return.

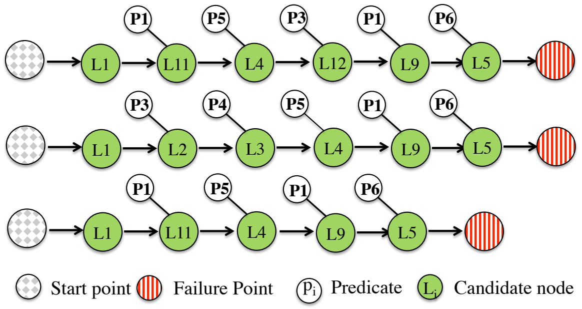
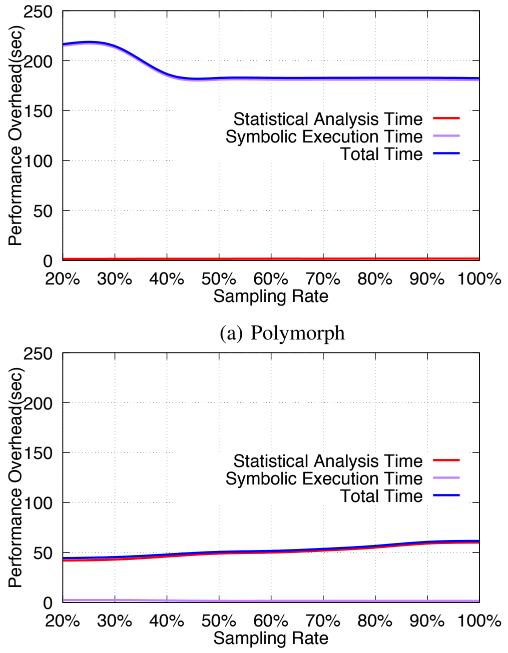


Fig. 9: Candidate paths for *Polymorph*

of internal and external calls. By reviewing the log files for thttpd, we observed that there are hundreds of function calls from the string injection point, *handle_read()* to the vulnerability site *defang()*. Among the large number of paths spawned by KLEE, only a few of them (less than 1%) reach the execution point at *defang()*, while a majority of explored paths are diverted from the real vulnerable path. Second, the *defang()* function involves a tight loop which includes an inner switch statement. Without additional constraints for the input string, KLEE itself will need to explore all the branch possibilities (every loop iteration and case statement). This will spawn even more states and quickly result in an explosion of states during exploration. For $StatSym$, with 30% sampling rate, the statistical analysis module finds a skeleton path consisting of 34 nodes along with 7 detours. Each of the instrumented location is a key execution point leading to *defang()*. Our statistical module identifies a number of high score predicates including *dfstr* in *defang* function as well as other variables and functions that are related to the *dfstr* variable. A total of thirteen candidate paths are constructed by joining the skeleton and detours. The length of the candidate paths range from 34 to 190 nodes. The first candidate path used to guide symbolic execution has 190 nodes. This path is generated by adding all the detours to the skeleton path in a randomly-chosen order. The symbolic execution module was not able to successfully explore along this candidate path, as the search failed after exceeding the hop threshold $\tau = 10$ (i.e., after exploring all matching execution paths that are no more than τ nodes different) at node *send_response():enter* with *node index 71*. This candidate path is marked as infeasible. With StatSym support, we are able to explore function *defang()* fairly quickly using the guidance from a candidate path identified in the second round. The predicate associated with this candidate node ($len(str) > 999.5$) further helps prune the search space for KLEE because additional forks of states relating to every loop iteration and switch-case branches can be eliminated for strings with shorter lengths. Overall, $StatSym$ used two candidate paths to finally discover the vulnerable path for thttpd.

*3) **CTree and Grep**:* The buffer overflow bug in CTree is triggered when an environment variable named $stonesoup\_stack\_buffer\_64$ with length over 64 bytes is read by CTree, which overflows a fixed stack buffer of size 64 in the function *initlinedraw()*. When using $StatSym$ to analyze CTree, the statistical analysis module is able to construct a candidate path that starts with the $main()$ program entry point, and ends at the exit point of $initlinedraw()$ function. The statistical analysis module generates the predicate for node $stonesoup\_read\_taint()\_leave$ showing that a string variable named $stonesoup\_tainted\_buff$ has length longer than 306.5 on vulnerable path. Note that the predicate gives a sufficient condition that lead to the vulnerability path since the actual length of the stack buffer is 64. This greatly reduces the required search space for KLEE during its symbolic execution. For Grep, the code injection mechanism by STONESOUP is similar to CTree. Our analysis shows that our approach constructs useful predicates that lead to vulnerable path discovery.

D. Sensitivity to Sampling



(a) Polymorph

(b) CTree

Fig. 10: Time breakdown for statistical analysis and symbolic execution modules of $StatSym$ to analyze *polymorph* and *CTree* when using different levels of sampling rate.

As shown in prior work [9], program logging with probabilistic sampling is required in practice to mitigate the runtime overhead. We evaluate the effects of reduced sampling rate using Fjalar [34]. While random sampling has advantages of reducing performance overhead, it increases the probability of false alarms in statistical analysis, leading to potentially lower accuracy in predicate and candidate path construction.

We conduct experiments, and show our results on polymorph and CTree applications to understand the implication of partial logs generated at different sampling rates. As sampling rate varies from 20% to 100%, we collect 200 log files from runtime program sampling and input the logs to $StatSym$. In Figure 10a and Figure 10b, we show the time required by

StatSym's statistical analysis module and symbolic execution module, respectively. Note that in all these experiments, *StatSym* is able to discover the vulnerable path. In particular, as sampling rate increases from 20% to 100%, the time spent by statistical analysis increases from 1.6 seconds to 1.9 seconds in polymorph and from 43.2 seconds to 58.7 seconds in CTree because of the larger sized log files.

As more program profile information is logged at runtime and becomes available for statistical analysis, the accuracy of results improve (i.e., increasing the probability of finding the vulnerable path). This is able to trim down the search space for symbolic executor significantly. Thus, the time spent by symbolic execution module to discover the vulnerable path decreases from 213.0 to 179.5 for polymorph and from 2.4 to 1.6 for CTree. These experimental results illustrate an interesting trade-off between more accurate inference (requiring increased runtime information and overhead) and larger search space that needs to be explored by symbolic execution. We note that with smarter techniques to implement partial logging, we are likely to achieve the same effectiveness for *StatSym* with even lower sampling rates.

VIII. Related Work

Statistical analysis has been employed to construct bug-related predicates [9], [10], [11], [37] to facilitate testing and debugging. In practice, this approach relies on partial logging (often through random sampling) of target programs to amortize monitoring overheads [9]. Prior works have studied predicate construction algorithms for bug localization, identifying multiple bugs [11], [38], [10]. Although these techniques can assist testing and debugging, their outputs still require manual analysis. Another line of work builds bug detection models using machine learning techniques, such as mining control-flow graphs [39] and hidden Markov model for anomaly detection [40], and modified support vector machine [41]. Different from those works, *StatSym* focuses on automated vulnerable program path diagnosis and debugging.

Symbolic execution, model checking and tainting are well studied techniques for formal verification, static analysis or test generation [13], [14], [42], [17], [43], [44], [45]. In [45], symbolic execution and symbolic reachability analysis are combined to improve the effectiveness of analyzing branches. This is further improved by fitness-guided symbolic execution [46], partial or directed symbolic execution [18], [19], [47], complex path constraints solving techniques [48], [49], [50], [51], [52], and concolic execution to generate concrete inputs for paths coverage [53], [54]. Although they are able to provide formal verification of program vulnerabilities, these techniques suffer from prohibitive overheads especially when analyzing large software programs that contain an exponential number of potential execution paths.

Li et al. [31], [30] leverage program runtime information for program debugging and security enhancement. Jin el al. [20] propose a bug synthesis tool that reproduces the observed field failures using execution data collected from users. However, this approach can have some practical limitations where logging entire (or most of) call sequences at runtime is infeasible. In this paper, by taking advantage of incomplete logging at a very low sampling rate, *StatSym* is able to minimize the runtime overhead. Crameri et al. [19] propose a technique that utilizes branch selection history to guide symbolic execution for debugging. This approach only offers a *local* view of program execution (at each individual branch) and thus are less effective in identifying the entire vulnerable path. In contrast, *StatSym* employs predicate and candidate path construct for a *global* search and is semantically more powerful. For example, consider the example of a loop that is branch-heavy: a branch direction for the loop only implies a direction choice for KLEE in this iteration, however, a statistically generated predicate that governs the number of iterations will be able to guide KLEE more effectively in reducing its overall search space.

IX. Conclusion

In this paper, we proposed *StatSym*, a novel framework for vulnerable path discovery, which harnesses the scalability of statistical analysis and the rigorousness of symbolic execution. Program runtime information is analyzed using the *StatSym* statistical analysis module to construct predicates and identify candidate vulnerable paths. Our statistical-guided symbolic executor leverages the paths to search vulnerable paths in a prioritized manner. We evaluated *StatSym* on four applications *polymorph* from Bugbench, *CTree*, *Grep* from NIST STONESOUP benchmarks and *thttpd*. Our results show *StatSym* has a **15×** speedup to find vulnerable paths compared to the pure symbolic execution - KLEE, and is able to correctly identify the vulnerabilities for all applications even when a pure symbolic executor fails in three out of the four applications.

Acknowledgment

This work was supported by the US Office of Naval Research (ONR) under Award N00014-15-1-2210. Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors, and do not necessarily reflect those of ONR.

References

[1] M. C. Libicki, L. Ablon, and T. Webb, *The Defenders Dilemma: Charting a Course Toward Cybersecurity*. Rand Corporation, 2015.

[2] P. Akulavenkatavara, J. Girouard, and E. Ratliff, "Mitigating Malicious Exploitation of A Vulnerability in A Software Application by Selectively Trapping Execution along A Code Path," 2010. US Patent 7,845,006.

[3] J. S. Mertoguno, "Human Decision Making Model for Autonomic Cyber Systems," *International Journal on Artificial Intelligence Tools*, 2014.

[4] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang, "R2: An Application-level Kernel for Record and Replay," in *USENIX Conference on Operating Systems Design and Implementation*, 2008.

[5] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "DoublePlay: Parallelizing Sequential Logging and Replay," *ACM Transactions on Computer Systems*, 2012.

[6] D. Subhraveti and J. Nieh, "Record and Transplay: Partial Checkpointing for Replay Debugging across Heterogeneous Systems," in *ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, 2011.

[7] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: Enabling Intrusion Analysis through Virtual-machine Logging and Replay," *ACM SIGOPS Operating Systems Review*, 2002.

[8] R. McNally, K. Yiu, D. Grove, and D. Gerhardy, "Fuzzing: The State of the Art," tech. rep., DTIC Document, 2012.

[9] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug Isolation via Remote Program Sampling," *ACM SIGPLAN Notices*, 2003.

[10] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical Debugging Using Compound Boolean Predicates," in *International Symposium on Software Testing and Analysis*, ACM, 2007.

[11] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable Statistical Bug Isolation," *ACM SIGPLAN Notices*, 2005.

[12] H. Xue, Y. Chen, F. Yao, Y. Li, T. Lan, and G. Venkataramani, "SIMBER: Eliminating Redundant Memory Bound Checks via Statistical Inference," in *International Conference on ICT Systems Security and Privacy Protection-IFIP SEC*, Springer, 2017.

[13] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *USENIX Conference on Operating Systems Design and Implementation*, 2008.

[14] I. Doudalis, J. Clause, G. Venkataramani, M. Prvulovic, and A. Orso, "Effective and Efficient Memory Protection Using Dynamic Tainting," *IEEE Transactions on Computers*, vol. 61, pp. 87–100, 2012.

[15] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel Symbolic Execution for Automated Real-world Software Testing," in *European Conference on Computer Systems*, ACM, 2011.

[16] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "MemTracker: An Accelerator for Memory Debugging and Monitoring," *ACM Transactions on Architecture and Code Optimization*, vol. 6, no. 2, pp. 5:1–5:33, 2009.

[17] J. Shen, G. Venkataramani, and M. Prvulovic, "Tradeoffs in Fine-grained Heap Memory Protection," in *Workshop on Architectural and System Support for Improving Software Dependability*, ACM, 2006.

[18] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed Symbolic Execution," in *International Static Analysis Symposium*, Springer, 2011.

[19] O. Crameri, R. Bianchini, and W. Zwaenepoel, "Striking A New Balance between Program Instrumentation and Debugging Time," in *European Conference on Computer Systems*, 2011.

[20] W. Jin and A. Orso, "BugRedux: Reproducing Field Failures for Inhouse Debugging," in *International Conference on Software Engineering*, IEEE Press, 2012.

[21] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving Software Diagnosability via Log Enhancement," *ACM Transactions on Computer Systems*, vol. 30, no. 1, pp. 4:1–4:28, 2012.

[22] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be Conservative: Enhancing Failure Diagnosis with Proactive Logging," in *USENIX Conference on Operating Systems Design and Implementation*, 2012.

[23] ACME Lab, "Thttpd." http://www.acme.com/software/thttpd/.

[24] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "BugBench: A Benchmark for Evaluating Bug Detection Tools," in *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[25] NIST, "IARPA STONESOUP Phase 3." https://samate.nist.gov/SARD/testsuite.php.

[26] J.-C. Laprie, "Dependable Computing: Concepts, Limits, Challenges," in *International Symposium On Fault-Tolerant Computing*, 1995.

[27] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When Good Instructions go bad: Generalizing Return-oriented Programming to RISC," in *ACM SIGSAC Conference on Computer and Communications Security*, 2008.

[28] CVE, "Vulnerability of thttpd in defang function." http://www.cvedetails.com/cve/2003-0899.

[29] Y. Zheng and X. Zhang, "Path Sensitive Static Analysis of Web Applications for Remote Code Execution Vulnerability Detection," in *International Conference on Software Engineering*, IEEE, 2013.

[30] Y. Li, F. Yao, T. Lan, and G. Venkataramani, "POSTER: Semantics-Aware Rule Recommendation and Enforcement for Event Paths," in *International Conference on Security and Privacy in Communication Systems*, pp. 572–576, Springer, 2015.

[31] Y. Li, F. Yao, T. Lan, and G. Venkataramani, "SARRE: Semantics-Aware Rule Recommendation and Enforcement for Event Paths on Android," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 12, pp. 2748–2762, 2016.

[32] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," in *International Conference on Very Large Data Bases*, 1994.

[33] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[34] P. J. Guo, "A Scalable Mixed-level Approach to Dynamic Analysis of C and C++ Programs," Master's thesis, MIT, 2006.

[35] "Verisec Suite." https://se.cs.toronto.edu/index.php/Verisec_Suite.

[36] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan, "DASE: Document-assisted Symbolic Execution for Improving Automated Software Testing," in *International Conference on Software Engineering*, IEEE, 2015.

[37] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: Statistical Model-based Bug Localization," *ACM SIGSOFT Software Engineering Notes*, 2005.

[38] S. Wang, F. Khomh, and Y. Zou, "Improving Bug Localization Using Correlations in Crash Reports," in *International Conference on Mining Software Repositories*, IEEE, 2013.

[39] X. Shu, D. Yao, and N. Ramakrishnan, "Unearthing Stealthy Program Attacks Buried in Extremely Long Execution Paths," in *ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[40] K. Xu, K. Tian, D. D. Yao, and B. G. Ryder, "A Sharper Sense of Self: Probabilistic Reasoning of Program Behaviors for Anomaly Detection with Context Sensitivity," in *International Conference on Dependable Systems & Networks*, IEEE/IFIP, 2016.

[41] Z. Gu, K. Pei, Q. Wang, L. Si, X. Zhang, and D. Xu, "LEAPS: Detecting Camouflaged Attacks with Statistical Learning Guided by Program Analysis," in *International Conference on Dependable Systems & Networks*, IEEE/IFIP, 2014.

[42] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable Accelerator for Dynamic Taint Propagation," in *International Symposium on High Performance Computer Architecture*, IEEE, 2008.

[43] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code," in *SIGOPS Operating Systems Review*, ACM, 2001.

[44] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking System Rules Using System-specific, Programmer-written Compiler Extensions," in *USENIX Conference on Operating Systems Design and Implementation*, 2000.

[45] M. Baluda, G. Denaro, and M. Pezze, "Bidirectional Symbolic Analysis for Effective Branch Testing," *IEEE Transactions on Software Engineering*, 2015.

[46] T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "Fitness-guided Path Exploration in Dynamic Symbolic Execution," in *International Conference on Dependable Systems & Networks*, IEEE/IFIP, 2009.

[47] C. Zamfir and G. Candea, "Execution Synthesis: A Technique for Automated Software Debugging," in *European Conference on Computer Systems*, ACM, 2010.

[48] A. Aquino, F. A. Bianchi, M. Chen, G. Denaro, and M. Pezzè, "Reusing Constraint Proofs in Program Analysis," in *International Symposium on Software Testing and Analysis*, ACM, 2015.

[49] P. A. Abdulla, M. F. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, "Norn: An SMT Solver for String Constraints," in *International Conference on Computer Aided Verification*, 2015.

[50] J. D. Scott, P. Flener, and J. Pearson, "Constraint Solving on Bounded String Variables," in *International Conference on Integration of AI and OR Techniques in Constraint Programming*, Springer, 2015.

[51] X. Xie, Y. Liu, W. Le, X. Li, and H. Chen, "S-looper: Automatic Summarization for Multipath String Loops," in *International Symposium on Software Testing and Analysis*, ACM, 2015.

[52] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended Symbolic Execution on Binary Programs," in *International Symposium on Software Testing and Analysis*, ACM, 2009.

[53] J. Zhang, X. Chen, and X. Wang, "Path-oriented Test Data Generation Using Symbolic Execution and Constraint Solving Techniques," in *International Conference on Software Engineering and Formal Methods*, IEEE, 2004.

[54] P. Dinges and G. Agha, "Targeted Test Input Generation Using Symbolic Concrete Backward Execution," in *International Conference on Automated Software Engineering*, ACM/IEEE, 2014.