Fuzzing Guided by Bayesian Program Analysis

Published: 08 January 2026
Accepted: 06 November 2025
Received: 10 July 2025

Fuzzing Guided by Bayesian Program Analysis

YIFAN ZHANG, Peking University, China
XIN ZHANG*, Peking University, China

We propose a novel approach that leverages Bayesian program analysis to guide large-scale target-guided greybox fuzzing (LTGF). LTGF prioritizes program locations (targets) that are likely to contain bugs and applies directed mutation towards high-priority targets. However, existing LTGF approaches suffer from coarse and heuristic target prioritization strategies, and lack a systematic design to fully exploit feedback from the fuzzing process. We systematically define this prioritization process as the reachable fuzzing targets problem. Bayesian program analysis attaches probabilities to analysis rules and transforms the analysis results into a Bayesian model. By redefining the semantics of Bayesian program analysis, we enable the prediction of whether each target is reachable by the fuzzer, and dynamically adjust the predictions based on fuzzer feedback. On the one hand, Bayesian program analysis builds Bayesian models based on program semantics, enabling systematic and fine-grained prioritization. On the other hand, Bayesian program analysis systematically learns feedback from the fuzzing process, making its guidance adaptive. Moreover, this combination extends the application of Bayesian program analysis from alarm ranking to fully automated bug discovery. We implement our approach and evaluate it against several state-of-the-art fuzzers. On a suite of real-world programs, our approach discovers 3.25× to 13× more unique bugs compared to baselines. In addition, our approach identifies 39 previously unknown bugs in well-tested programs, 30 of which have been assigned CVEs.

CCS Concepts: • **Security and privacy** → **Software security engineering**.

Additional Key Words and Phrases: Greybox Fuzzing, Target Prioritization, Static Analysis, Bayesian Network

ACM Reference Format:
Yifan Zhang and Xin Zhang. 2026. Fuzzing Guided by Bayesian Program Analysis. *Proc. ACM Program. Lang.* 10, POPL, Article 17 (January 2026), 31 pages. https://doi.org/10.1145/3776659

1 Introduction

Large-scale target-guided greybox fuzzing (LTGF) [5, 33, 49, 50] is one of the most effective methods in recent years for discovering unknown bugs in programs. LTGF uses multiple locations in a program as a target set (usually derived from static analysis results and sanitizers [38]) and continuously performs target prioritization during the fuzzing process. For high-priority targets, LTGF leverages techniques from directed greybox fuzzing (DGF) [2] to mutate the seeds that are more likely to trigger bugs at these targets. As a result, LTGF can trigger more bugs within the same amount of time compared to conventional coverage-guided greybox fuzzing (CGF) [46].

However, the target prioritization in existing LTGF approaches is coarse and lacks systematic design. For example, in two recent LTGF works, FishFuzz [50] and Prospector [49], they prioritize targets that have been exercised fewer times by the generated test inputs. Specifically, for two targets l_1 and l_2, let $\textsc{Count}(l_1)$ and $\textsc{Count}(l_2)$ denote the number of generated test inputs that

*Corresponding author.

Authors' Contact Information: Yifan Zhang, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, yfzhang23@stu.pku.edu.cn; Xin Zhang, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, xin@pku.edu.cn.

ACM 2475-1421/2026/1-ART17
https://doi.org/10.1145/3776659

have exercised the location of l_1 and l_2, respectively. If $\textsc{Count}(l_1) < \textsc{Count}(l_2)$, these approaches will prioritize attempting to trigger vulnerabilities at l_1. Such heuristic-based prioritization is not only ad-hoc and coarse-grained, but also fails to fully leverage feedback from the fuzzing process, thereby restricting the effectiveness and full potential of LTGF.

To systematically optimize this process, we first formally define target prioritization as a reachable fuzzing targets problem. Specifically, we define the probability that each target program property can be reached by a directed fuzzer. We then formulate the reachable fuzzing targets problem as predicting whether the probability for each property to be reached by the fuzzer exceeds a given threshold. To address this problem, we propose a method based on Bayesian program analysis [26] to predict reachable fuzzing targets. Our method constructs a Bayesian model on top of static analysis derivations, learns from the fuzzer feedback as posterior information, and ultimately uses probabilistic inference to compute the probability that each target program property is reachable by the fuzzer.

Our approach offers two major advantages for guiding LTGF: (1) Compared to previous work [4, 11, 15, 20, 34, 47] on Bayesian program analysis, we fundamentally redefine the semantics of the Bayesian network. For each program property that is over-approximated from the program semantics, we redefine the prediction task from determining whether the property is satisfied by at least one input to determining whether the property is reachable by the fuzzer. This enables our approach to be *semantics-aware* and *fine-grained*. (2) Based on the semantics we defined for the Bayesian network, we systematically present a method to generalize fuzzer feedback as posterior information, which makes our approach *adaptive*.

In addition, all existing work [4, 11, 15, 20, 26, 34, 47] has focused on applying Bayesian program analysis to prioritize static analysis alarms to assist users in manual inspection. In this paper, we aim to extend the application of Bayesian program analysis to guiding fuzzing, enabling fully automated bug discovery and demonstrating greater practical utility.

We present Bayzzer, a framework that applies Bayesian program analysis to guide LTGF. The Bayesian program analysis continuously guides the fuzzer through multiple rounds of interaction. In each interaction round: (1) The Bayesian model calculates the probability that each target is reachable by the fuzzer and ranks them accordingly. The fuzzer then attempts to trigger bugs at the top-ranked targets. (2) For targets where bugs are successfully triggered, it indicates that such bugs are reachable for the fuzzer. The fuzzer gives positive feedback to the Bayesian model, increasing the probabilities associated with similar targets, allowing the fuzzer to trigger similar bugs more quickly. For targets where no bugs are triggered, it indicates that they are false positives over-approximated by static analysis, or that the fuzzer is currently not capable of triggering them (for instance, the target is only reachable when a strict condition like `if(x == 998244353)` is satisfied, which makes it hard to trigger during fuzzing). The fuzzer gives negative feedback to the Bayesian model, decreasing the probabilities of the associated targets, thereby avoiding wasted effort on such bugs. (3) As fuzzing progresses, the fuzzer accumulates more seeds and becomes increasingly capable. Some previously untriggerable bugs may become triggerable. At this point, the interaction process restarts, and all previous negative feedback is cleared.

We have implemented Bayzzer on top of the LTGF framework FishFuzz [50], and compared it against the following fuzzers: (1) Prospector [49], a state-of-the-art LTGF; (2) FishFuzz, the base framework of Bayzzer; (3) FunFuzz [45], a non-Bayesian static-analysis-based state-of-the-art CGF; (4) AFL++ [8], the base framework for all other fuzzers used in our evaluation. We use the same set of 24 real-world programs and evaluation metrics as Prospector. The results show that Bayzzer discovers 3.25×, 6.5×, 6.5×, and 13× more unique bugs compared to these baselines. Moreover, Bayzzer discovers 39 new bugs from well-tested programs, including latest versions of the above programs and popular open-source projects supported by OSS-Fuzz [10], 30 of which have been assigned CVEs.

Contributions. This paper makes the following contributions:

(1) We systematically define the reachable fuzzing targets problem, and by redefining the semantics of Bayesian program analysis, we enable both accurate prediction of this problem and generalization of fuzzer feedback.
(2) We propose a framework Bayzzer for leveraging Bayesian program analysis to systematically provide fine-grained, semantics-aware, and adaptive guidance to fuzzing.
(3) We extend the application scope of Bayesian program analysis from ranking static analysis alarms to guiding fuzzing for fully automated bug discovery.
(4) We show the effectiveness of Bayzzer on a suite of real-world programs. Bayzzer demonstrates stronger bug detection performance compared to the baselines.

2 Motivating Example

This section we illustrate the limitations of the coarse-grained target prioritization adopted by existing LTGF approaches, and demonstrate how Bayesian program analysis can provide semantics-aware, fine-grained, and adaptive guidance to address these issues through two case studies: (1) We begin with the discovery of CVE-2017-14409 [27] and CVE-2017-14410 [28] in `mp3gain-1.5.2`. This case study illustrates the core workflow of Bayesian program analysis and how it enables semantics-aware and fine-grained guidance for fuzzing. (2) We then present the discovery of CVE-2022-27941 [29] and CVE-2022-27942 [30] in `tcpreplay-4.4.0`. This case study shows how Bayesian program analysis can adaptively guide fuzzing to select more suitable targets based on the fuzzer's own capabilities.

2.1 A Taint Analysis for Detecting Memory Errors

In this subsection, we first introduce the code used in our first case study. Then, we apply a taint analysis written in Datalog to detect potential memory errors in the code. The generated alarms are used as targets in large-scale target-guided greybox fuzzing (LTGF). LTGF attempts to trigger potential vulnerabilities at these targets. This taint analysis also serves as the logical component of the Bayesian program analysis used in the subsequent subsections.

Figure 1 shows a simplified code fragment from `mp3gain-1.5.2` that contains CVE-2017-14409 and CVE-2017-14410. For clarity, we partly rewritten the code but keep the semantics related to the vulnerabilities. Beginning at Line 29, the function `get1bit()` reads one bit from the input MP3. The global structure `gr_infos` holds input metadata, and its field `mixed_block_flag` gets the value from `get1bit()`. Next, at Line 30, the code invokes `III_i_stereo` with `gr_info` as an argument. Later, `gr_infos->mixed_block_flag` contributes to computing `is_p` at Line 21. Finally, `is_p` is used as an array index at Line 22. Without bounds checking on `is_p`, manipulating `gr_infos->mixed_block_flag` can cause out-of-bounds access and a global buffer overflow. This vulnerability corresponds to CVE-2017-14410. CVE-2017-14409 has a similar cause. At Line 31, `III_dequantize_sample` is called with `gr_info`. Then, `gr_infos->mixed_block_flag` helps compute pointer `m` at Line 14, affecting `xrpnt` at Line 15. Finally, the lack of bounds checks allows the dereference of `xrpnt` at Line 16 to trigger a global buffer overflow by manipulating `gr_infos->mixed_block_flag`. Next, at Line 32, `getbits_fast` reads input and assigns it to `gr_infos->scalefac_compress`. Afterwards, the code invokes `III_get_scale_factors_1` and `III_get_scale_factors_2` with `gr_infos` at Line 33 and Line 34. At Line 3 and Line 9, the code uses `gr_infos->scalefac_compress` as an array index. Since `check_1` and `check_2` are invoked

```
1   static int III_get_scale_factors_1(struct gr_info_s
        * gr_infos, ...){
2     if(!check_1(gr_infos->scalefac_compress)) return
          -1;
3     int num0 = slen[0][gr_infos->scalefac_compress];
4     ...
5   }
6
7   static int III_get_scale_factors_2(struct gr_info_s
        * gr_infos, ...){
8     if(!check_2(gr_infos->scalefac_compress >> 1))
          return -1;
9     unsigned int slen = i_slen2[gr_infos->
          scalefac_compress >> 1];
10    ...
11  }
12
13  static int III_dequantize_sample(struct gr_info_s*
        gr_infos, ...){
14    register int* m = map[sfreq][gr_infos->
          mixed_block_flag];
15    real* xrpnt = ((real*)xr) + (*m++);
16    *xrpnt = ispow[y] * v; // CVE-2017-14409
17    ...
18  }
19
20  static void III_i_stereo(struct gr_info_s* gr_infos
        , ...){
21    int is_p = scalefac[sfb * 3 + lwin - gr_infos->
          mixed_block_flag];
22    real t1 = tabl1[is_p]; // CVE-2017-14410
23    ...
24  }
25
26  struct gr_info_s* gr_infos = ...
27
28  int main(){
29    gr_infos->mixed_block_flag = get1bit();
30    III_i_stereo(gr_infos, ...);
31    III_dequantize_sample(gr_infos, ...);
32    gr_infos->scalefac_compress = getbits_fast(4);
33    III_get_scale_factors_1(gr_infos, ...);
34    III_get_scale_factors_2(gr_infos, ...);
35    ...
36  }
```

Fig. 1. Simplified code fragment from `mp3gain-1.5.2` containing CVE-2017-14409 and CVE-2017-14410.

Input relations

$\mathsf{Input}(v)$:	The value of variable v comes from external input.
$\mathsf{Flow}(v_1, v_2)$:	There exists a data flow from v_1 to v_2.
$\mathsf{Memory}(v, s)$:	v involves in memory operations in statement s.

Output relations

$\mathsf{Taint}(v)$:	Variable v is tainted.
$\mathsf{Alarm}(s)$:	Statement s leads to a memory error.

Derivation rules

R_1 :	$\mathsf{Taint}(v)$:- $\mathsf{Input}(v)$.
R_2 :	$\mathsf{Taint}(v_2)$:- $\mathsf{Taint}(v_1), \mathsf{Flow}(v_1, v_2)$.
R_3 :	$\mathsf{Alarm}(s)$:- $\mathsf{Taint}(v), \mathsf{Memory}(v, s)$.

Fig. 2. A taint analysis in Datalog.

at Line 2 and Line 8, respectively, to check the array indices used in the subsequent line, it is ensured that the indices are within the valid range. Therefore, no out-of-bounds access occurs.

We use a taint analysis to generate alarms in the program. These alarms serve as the target set for LTGF to attempt to trigger vulnerabilities. Figure 2 shows the taint analysis written in Datalog. This analysis is context-insensitive but flow-sensitive. It treats all inputs as taint sources and propagates taint via potential data flows. A value is tainted if it can be directly controlled by an attacker through input. If a tainted variable is used in memory operations (e.g., array indexing or pointer arithmetic), the analysis raises an alarm. Figure 2 contains full definitions of input and output relations. The analysis uses three derivation rules: R_1 taints inputs, R_2 propagates taint, and R_3 handles memory operations with tainted values. All three rules over-approximate and can produce spurious facts. Specifically, R_1 and R_3 ignore how branches limit variable ranges, while

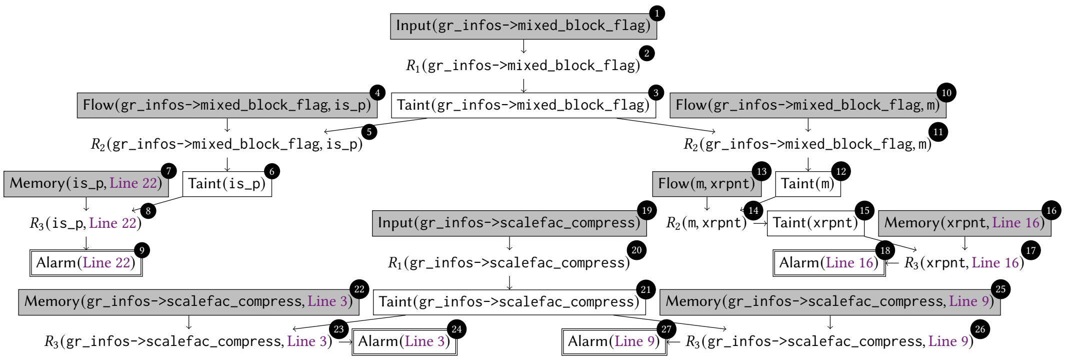
Input(gr_infos->mixed_block_flag)
R1(gr_infos->mixed_block_flag)
Flow(gr_infos->mixed_block_flag, is_p)
Taint(gr_infos->mixed_block_flag)
Flow(gr_infos->mixed_block_flag, m)
R2(gr_infos->mixed_block_flag, is_p)
R2(gr_infos->mixed_block_flag, m)
Memory(is_p, Line 22)
Taint(is_p)
Flow(m, xrpnt)
Taint(m)
R3(is_p, Line 22)
Input(gr_infos->scalefac_compress)
R2(m, xrpnt)
Taint(xrpnt)
Memory(xrpnt, Line 16)
Alarm(Line 22)
R1(gr_infos->scalefac_compress)
Alarm(Line 16)
R3(xrpnt, Line 16)
Memory(gr_infos->scalefac_compress, Line 3)
Taint(gr_infos->scalefac_compress)
Memory(gr_infos->scalefac_compress, Line 9)
R3(gr_infos->scalefac_compress, Line 3)
Alarm(Line 3)
Alarm(Line 9)
R3(gr_infos->scalefac_compress, Line 9)

Fig. 3. The derivation graph of the analysis in Figure 2 applying to the code fragment in Figure 1. Vertices with a bordered gray background represent input tuples. Vertices with a bordered white background represent output tuples, while vertices with a double-bordered white background represent alarms tuples. Vertices without borders represent rule instances. For brevity, we use a black-circled number at the top right corner of each vertex to indicate its identifier.

R_2 ignores context differences. A Datalog engine applies the derivation rules to input tuples and iterates until reaching a fixed point, where no new output tuples are produced.

We visualize the Datalog derivation as a directed graph for clarity. We refer to such a graph as the *derivation graph*. Figure 3 shows the derivation graph from Figure 2 applied to the code in Figure 1. The graph includes input tuples (bordered, grey), output tuples (bordered, white), and rule instances (no borders) generating the output tuples. A rule instance represents a single application of a derivation rule that produces a new output tuple. For instance, R_2(m, xrpnt) uses Taint(m) and Flow(m, xrpnt) to derive Taint(xrpnt). Specifically, vertices with double borders and white backgrounds denote alarm tuples. Alarm(Line 16) and Alarm(Line 22) match real bugs CVE-2017-14409 and CVE-2017-14410. Alarm(Line 3) and Alarm(Line 9) are false positives from static analysis over-approximation. Since the taint analysis only models whether variables are tainted by external input and does not model the specific values, it cannot determine that the bounds checks at Line 2 and Line 8 will prevent out-of-bounds array accesses.

If we adopt the existing LTGF target prioritization strategy, which prioritizes targets that have been exercised the fewest times by generated inputs, two main issues arise: (1) Because the number of times a focused target is exercised can rapidly increase, all targets tend to receive equal attention in the long run. As a result, this strategy cannot distinguish between false alarms such as Alarm(Line 3) and Alarm(Line 9), leading to significant time wasted on these false targets. (2) It fails to associate alarms. For example, triggering the vulnerabilities at Line 16 and Line 22 is closely related, but even after successfully triggering the vulnerability at Line 16, the strategy cannot leverage this information to focus on Alarm(Line 22). Next, we show how Bayesian program analysis addresses these two issues and helps the fuzzer prioritize more promising alarms.

2.2 Semantics-Aware and Fine-Grained Fuzzing Guidance via Bayesian Program Analysis

In this subsection, we use the previously introduced taint analysis as the logic for Bayesian program analysis, and show how it guides fuzzing to find CVE-2017-14409 and CVE-2017-14410 while minimizing the prioritization of false alarms. The core idea is that (1) Bayesian program analysis builds a semantics-based Bayesian network for *semantics-aware* guidance; and (2) Bayesian program analysis ranks fuzzing targets via systematic Bayesian inference. The ranking adapts to fuzzer

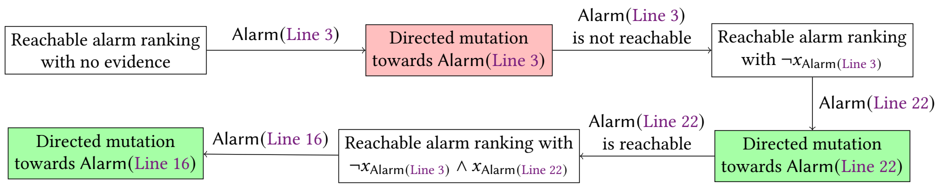
Reachable alarm ranking with no evidence
Alarm(Line 3)
Directed mutation towards Alarm(Line 3)
Alarm(Line 3) is not reachable
Reachable alarm ranking with $\neg x_{\text{Alarm(Line 3)}}$
Alarm(Line 22)
Directed mutation towards Alarm(Line 22)
Alarm(Line 22) is reachable
Reachable alarm ranking with $\neg x_{\text{Alarm(Line 3)}} \wedge x_{\text{Alarm(Line 22)}}$
Alarm(Line 16)
Directed mutation towards Alarm(Line 16)

Fig. 4. The full workflow of Bayesian program analysis guiding fuzzing for the code fragment in Figure 1.

feedback, avoiding false positives and revealing bugs with shared root causes, thereby enabling *fine-grained* guidance.

Returning to the case study: since R_1, R_2, and R_3 over-approximate, we assign probabilities to derivations to capture uncertainty. We simplify by assigning 0.9 probability to each rule. With this view, the derivation graph becomes a Bayesian network [17]. Each tuple or rule instance t maps to a Bernoulli variable x_t, which indicates whether it holds under the set of inputs that the fuzzer can generate. For example, Taint(xrpnt) and R_2(m, xrpnt) correspond to $x_{\texttt{Taint(xrpnt)}}$ and $x_{R_2\texttt{(m,xrpnt)}}$. Their dependencies follow the structure of the derivation graph. For each rule instance, we specify a corresponding conditional probability. We provide the explicit conditional probability distribution for this Bayesian network. For brevity, we use the identifiers at the top right corner of each vertex in Figure 3 to denote the corresponding vertex:

(1) **Input tuple facts:** For vertices $a \in \{1, 4, 7, 10, 13, 16, 19, 22, 25\}$, we have $\Pr(x_a) = 1$.
(2) **Over-approximated inferences of rule R_1:** For vertices $(a, b, c) \in \{(1, 2, 3), (19, 20, 21)\}$, we have:

$$\begin{aligned} \Pr(x_b \mid x_a) &= 0.9 & \Pr(\neg x_b \mid x_a) &= 0.1 \\ \Pr(x_b \mid \neg x_a) &= 0 & \Pr(\neg x_b \mid \neg x_a) &= 1 \\ \Pr(x_c \mid x_b) &= 1 & \Pr(\neg x_c \mid x_b) &= 0 \\ \Pr(x_c \mid \neg x_b) &= 0 & \Pr(\neg x_c \mid \neg x_b) &= 1 \end{aligned}$$

(3) **Over-approximated inferences of rules R_2 and R_3:** For vertices

$$(a, b, c, d) \in \begin{Bmatrix} (3, 4, 5, 6), (3, 10, 11, 12), (12, 13, 14, 15), (6, 7, 8, 9), \\ (15, 16, 17, 18), (21, 22, 23, 24), (21, 25, 26, 27) \end{Bmatrix}$$

we have:

$$\begin{aligned} \Pr(x_c \mid x_a \wedge x_b) &= 0.9 & \Pr(\neg x_c \mid x_a \wedge x_b) &= 0.1 \\ \Pr(x_c \mid \neg x_a \vee \neg x_b) &= 0 & \Pr(\neg x_c \mid \neg x_a \vee \neg x_b) &= 1 \\ \Pr(x_d \mid x_c) &= 1 & \Pr(\neg x_d \mid x_c) &= 0 \\ \Pr(x_d \mid \neg x_c) &= 0 & \Pr(\neg x_d \mid \neg x_c) &= 1 \end{aligned}$$

This conditional probability distribution links Bernoulli variables via semantics-based probabilistic relationships.

We present the complete workflow of Bayesian program analysis guiding fuzzing to ultimately trigger the target vulnerability in Figure 4. Next, we provide a detailed description of this process. We compute the probability of each alarm being true by performing marginal inference [32] on the Bayesian network derived from Figure 3, as shown in Table 1a. As Figure 1 covers only part of the program, the taint analysis also reports alarms in other parts, including both true and false ones. To simplify presentation, we include only the four relevant alarms (Line 3, Line 9, Line 22, Line 16) and exclude the rest.

Table 1. The probability that each alarm can be reached by the fuzzer, based on the Bayesian network transformed from Figure 3, before and after fuzzer feedback on Line 3 and Line 22.

(a) $\Pr\left(x_{\mathsf{Alarm}(s)}\right)$

Rank	Prob.	Target s
1	0.810	Line 3
2	0.810	Line 9
3	0.729	Line 22
...	...	...
23	0.656	Line 16
...	...	...
...	...	...

(b) $\Pr\left(x_{\mathsf{Alarm}(s)} \mid \neg x_{\mathsf{Alarm}(\text{Line 3})}\right)$

Rank	Prob.	Target s
1	0.729	Line 22
...	...	...
22	0.656	Line 16
...	...	...
233	0.426	Line 9
...	...	...
-	0.000	Line 3

(c) $\Pr\left(x_{\mathsf{Alarm}(s)} \mid \neg x_{\mathsf{Alarm}(\text{Line 3})} \wedge x_{\mathsf{Alarm}(\text{Line 22})}\right)$

Rank	Prob.	Target s
-	1.000	Line 22
1	0.729	Line 16
...	...	...
232	0.426	Line 9
...	...	...
...	...	...
-	0.000	Line 3

Alarm(Line 3) and Alarm(Line 9) have the highest probabilities since they involve fewer imprecise derivations. We select the highest-probability alarm, assumed to be Line 3, as the fuzzing target. The fuzzer uses directed fuzzing to mutate inputs toward the target, aiming to trigger the potential bug. As Line 3 is a false positive, the fuzzer fails to trigger the bug. After a period of mutation, the fuzzer feeds back that the bug is not triggerable. The Bayesian network adds negative evidence to $x_{\mathsf{Alarm}(\text{Line 3})}$ and updates the probabilities of other alarms accordingly, as shown in Table 1b. Because $x_{\mathsf{Alarm}(\text{Line 9})}$ is linked to $x_{\mathsf{Alarm}(\text{Line 3})}$ in the network, its probability also drops, leading to a lower rank. This helps the fuzzer focus on higher-ranked alarms and avoid wasting effort on false positives from the same root cause.

The next highest-probability alarm is Alarm(Line 22), chosen as the next fuzzing target. This alarm corresponds to CVE-2017-14410. As the bug lacks strict conditions, the fuzzer triggers it quickly. After triggering, the fuzzer provides positive feedback to the Bayesian network. The network adds positive evidence to $x_{\mathsf{Alarm}(\text{Line 22})}$ and updates alarm probabilities, as shown in Table 1c. Because $x_{\mathsf{Alarm}(\text{Line 16})}$ is linked to $x_{\mathsf{Alarm}(\text{Line 22})}$, its probability rises and becomes the top-ranked alarm. The fuzzer then targets Line 16. This alarm corresponds to CVE-2017-14409, which shares the same root cause as CVE-2017-14410. Both derive from the manipulation of `gr_infos->mixed_block_flag`. So the fuzzer quickly triggers this bug as well. Thus, the Bayesian network helps the fuzzer efficiently find both CVE-2017-14409 and CVE-2017-14410.

2.3 Adaptive Fuzzing Guidance via Bayesian Program Analysis

In this subsection, we present the code for our second case study and show how to improve the previous approach to guide fuzzing toward CVE-2022-27941 and CVE-2022-27942. The core insight is that the fuzzer becomes stronger over time. Here, "strong" refers to the **increased number of seeds collected**, which **enhances the fuzzer's targeted triggering capability**. In early stages, a weak fuzzer may miss bugs at some targets. We let the fuzzer give negative feedback, and the Bayesian model steers it away from similar hard-to-trigger patterns. Once the fuzzer improves, we restart interaction and clear earlier negative feedback. The stronger fuzzer may now reach bugs that were missed before. When it finds such bugs, the Bayesian model again guides prioritization of related targets. This enables *adaptive* guidance by adjusting feedback types to match the fuzzer's evolving capability.

Figure 5 shows simplified code from `tcpreplay-4.4.0` with CVE-2022-27941 and CVE-2022-27942. As in the previous case, the code is simplified for clarity. Beginning at Line 18, `safe_pcap_next()` reads a packet from the input PCAP file and assigns it to `pktdata`. Then, the code invokes `get_l2len_protocol()` with `pktdata` at Line 20. At Line 8, the code checks the first three bytes of

```
1   int parse_mpls(u_char* pktdata, uint32_t* l2len,
        ...){
2     struct tcpr_mpls_label* mpls_label = pktdata + *
          l2len;
3     bool bos = mpls_label->entr; // CVE-2022-27942
4     ...
5   }
6
7   int get_l2len_protocol(u_char* pktdata, uint32_t*
        l2offset, ...){
8     if(memcmp(pktdata, "MGC", 3)) return -1;
9     *l2offset = *((uint16_t*)&pktdata[4]);
10    eth_hdr_t* eth_hdr = (eth_hdr_t*)(pktdata + *
          l2offset);
11    uint32_t l2_net_off = sizeof(*eth_hdr) + *
          l2offset;
12    uint16_t ether_type = eth_hdr->ether_type; // CVE
          -2022-27941
13    parse_mpls(pktdata, &l2_net_off);
14    ...
15  }
16
17  int main(){
18    u_char* pktdata = safe_pcap_next(...);
19    uint32_t l2offset;
20    get_l2len_protocol(pktdata, &l2offset, ...);
21    ...
22  }
```

Fig. 5. Simplified code fragment from `tcpreplay-4.4.0` containing CVE-2022-27941 and CVE-2022-27942.

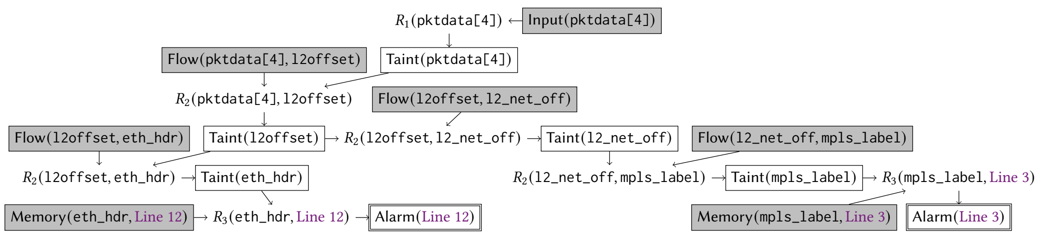

Fig. 6. The derivation graph of the analysis in Figure 2 applying to the code fragment in Figure 5. Vertices with a bordered gray background represent input tuples. Vertices with a bordered white background represent output tuples, while vertices with a double-bordered white background represent alarms tuples. Vertices without borders represent rule instances.

`pktdata`. Execution continues only if the bytes match magic number `MGC`; otherwise, it exits early. At Line 9, `pktdata[4]` is assigned to `l2offset`, later used to compute `eth_hdr` and `l2_net_off` at Line 10 and Line 11. At Line 12, the code accesses a field through `eth_hdr`. Insufficient bounds checks may cause this access to trigger a heap buffer overflow. This corresponds to CVE-2022-27941. Next, the code invokes `parse_mpls()` at Line 13 with `l2_net_off` as `l2len`. At Line 2, `l2len` is used to compute `mpls_label`. Finally, at Line 3, a field is accessed through `mpls_label`. Again, lack of bounds checks may lead to a heap buffer overflow. This corresponds to CVE-2022-27942.

We apply the taint analysis from Figure 2 to Figure 5, obtaining the derivation graph shown in Figure 6. The two alarms, Alarm(Line 12) and Alarm(Line 3), correspond to CVE-2022-27941 and CVE-2022-27942, respectively. We convert the graph into a Bayesian network using the same conditional probability distribution settings as in the previous case study. We present the complete workflow of Bayesian program analysis guiding fuzzing to ultimately trigger the target vulnerability in Figure 7. Next, we provide a detailed description of this process.

We compute the marginal probabilities of each alarm using Bayesian inference. The resulting ranking is shown in Table 2a. As before, other alarms are omitted for clarity. We choose the highest-ranked alarm at Line 12 as the fuzzing target. At the start of fuzzing, the magic number check at Line 8 stops execution unless the input begins with `MGC`. Greybox fuzzing lacks semantic awareness, so the chance of producing an input with the correct `MGC` prefix is very low. As a

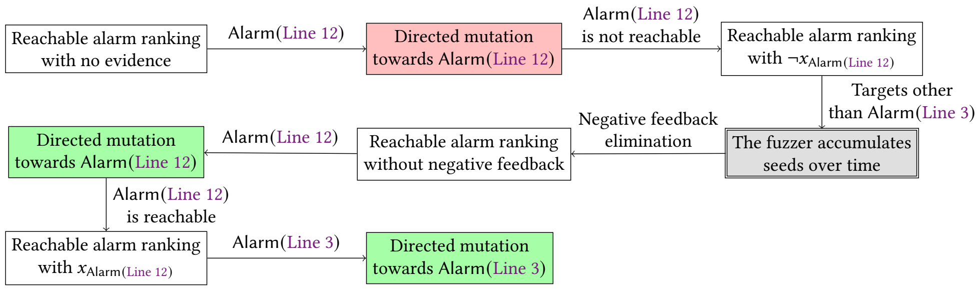
Reachable alarm ranking with no evidence
Alarm(Line 12)
Directed mutation towards Alarm(Line 12)
Alarm(Line 12) is not reachable
Reachable alarm ranking with $\neg x_{\text{Alarm(Line 12)}}$
Targets other than Alarm(Line 3)
The fuzzer accumulates seeds over time
Negative feedback elimination
Reachable alarm ranking without negative feedback
Alarm(Line 12)
Directed mutation towards Alarm(Line 12)
Alarm(Line 12) is reachable
Reachable alarm ranking with $x_{\text{Alarm(Line 12)}}$
Alarm(Line 3)
Directed mutation towards Alarm(Line 3)

Fig. 7. The full workflow of Bayesian program analysis guiding fuzzing for the code fragment in Figure 5.

Table 2. The probability that each alarm can be reached by the fuzzer, as computed from the Bayesian network derived from Figure 6, before and after incorporating feedback on Line 12 from fuzzers with different capabilities.

(a) $\Pr\left(x_{\text{Alarm}(s)}\right)$

Rank	**Prob.**	**Target** s
1	0.656	Line 12
2	0.590	Line 3
...	...	...
...	...	...

(b) $\Pr\left(x_{\text{Alarm}(s)} \mid \neg x_{\text{Alarm(Line 12)}}\right)$

Rank	**Prob.**	**Target** s
...	...	...
233	0.326	Line 3
...	...	...
-	0.000	Line 12

(c) $\Pr\left(x_{\text{Alarm}(s)} \mid x_{\text{Alarm(Line 12)}}\right)$

Rank	**Prob.**	**Target** s
-	1.000	Line 12
1	0.729	Line 3
...	...	...
...	...	...

result, the fuzzer cannot trigger the real bug due to its limited mutation ability. Later, the fuzzer reports to the Bayesian network that the bug at Line 12 is not triggered. The network adds negative evidence to the alarm and updates the probabilities of the other alarms, as shown in Table 2b. Because $x_{\text{Alarm(Line 3)}}$ is linked to $x_{\text{Alarm(Line 12)}}$, its probability also drops. The paths to both alarms are similar, and attempts to reach Line 3 are also blocked by the magic number check. This helps the fuzzer focus on other targets and avoid wasting effort on unreachable bugs at the current stage.

Over time, the fuzzer collects more seeds and becomes more effective at triggering bugs. Previously unreachable bugs may now become triggerable. We restart the interaction and clear prior negative feedback, as those bugs may now be reachable. The updated ranking matches Table 2a, so we again select Line 12 as the fuzzing target. Through prolonged mutation, the fuzzer eventually generates an input that passes the magic number check. Once past the check, the bug at Line 12 is easy to trigger and is quickly exposed. The fuzzer reports success to the Bayesian network, which adds positive evidence to Alarm(Line 12). The Bayesian network then updates the alarm probabilities based on the new posterior, as shown in Table 2c.

Because $x_{\text{Alarm(Line 3)}}$ is linked to $x_{\text{Alarm(Line 12)}}$, its probability increases and it becomes top-ranked. The fuzzer then prioritizes Line 3. This corresponds to CVE-2022-27942, which shares the root cause with CVE-2022-27941. Both derive from manipulating `pktdata[4]` in the input. As the fuzzer can now pass the magic check, it quickly triggers this bug as well.

Next, we formalize the definition of the problem we address and our proposed solution in Section 3, and present the detailed design of our Bayzzer framework in Section 4.

3 The Reachable Fuzzing Targets Problem and Our Solution

First of all, we present the problem that this paper aims to solve: how to optimize target prioritization so that large-scale target-guided greybox fuzzing (LTGF) can trigger more vulnerabilities. We define a subproblem: how to predict targets that are highly likely to be reached by a fuzzer. By solving this subproblem, we achieve optimization of the original problem. We begin by providing a formal definition of this problem in Section 3.1, We give the formal definitions of Datalog-based program analysis in Section 3.2. Finally, we formally define our solution to the problem in Section 3.3: we construct a Bayesian network to predict the probability that each target is reachable by the fuzzer.

3.1 Reachable Fuzzing Targets Problem

Large-scale target-guided greybox fuzzing (LTGF) is a complex process composed of multiple basic fuzzing processes and consists of two stages: exploration and exploitation. The exploration stage aims to cover as much code as possible, while the exploitation stage focuses on triggering vulnerabilities in the target set. We first define the notion of targets, then formalize the basic fuzzing process, and on this basis, define LTGF. Finally, we introduce the subproblem of the reachable fuzzing targets problem from the exploitation stage of LTGF.

We first provide a definition for the targets that can be reached by a fuzzer. In fact, a target typically represents a program property, such as "`p` points to `q`", "`x` is controlled by external input", or "an array out-of-bounds error occurs at Line 17", and so on. A program property may only hold under specific inputs, and we formalize this notion below.

Definition 3.1 (program property and program input). We define $\mathbf{P}$ as the set of program properties and $\mathbf{S}$ as the set of program inputs. The function $\textsc{Hold}(p, s) : \mathbf{P} \times \mathbf{S} \rightarrow \{\texttt{true}, \texttt{false}\}$ indicates whether the program property $p \in \mathbf{P}$ holds when executing the program with input $s \in \mathbf{S}$.

Next, we define the process of fuzzing. Given an initial set of inputs, a time budget, and a strategy (which indicates which seed to select for mutation at each iteration), an input is selected according to the strategy, and then transformed into a new input using various mutator operators. The new input is then used to test the program. We formally define this process as follows.

Definition 3.2 (fuzzing). We define a *strategy* $G(S, n) : \mathcal{P}(\mathbf{S}) \times \mathbb{R} \rightarrow \mathbf{S} \cup \{\perp\}$, which determines, for the current input set S, and the remaining time n, which input is selected for the next mutation. If G returns $\perp$, it indicates that the remaining time is exhausted and no further mutation is necessary.

Let $M = \mathbf{S} \times \mathbf{S} \times [0, 1] \times \mathbb{R}$ represent the probability and the total consumed time (mutation and execution new input) of each input mutating into another new input after one mutation. For any $s \in \mathbf{S}$, we require that $\sum_{s' \in \mathbf{S},\ (s,s',r,n) \in M} r = 1$.

Given a strategy G, a total time budget $N \in \mathbb{R}$ and an initial input set $S_0 \subseteq \mathbf{S}$, the process of fuzzing is formally defined as follows:

(1) Initialize the current input set $S_{\text{cur}} \leftarrow S_0$ and the remaining time $n_{\text{cur}} \leftarrow N$.
(2) At each step, apply the strategy G. If $G(S_{\text{cur}}, n_{\text{cur}}) = \perp$, terminate the fuzzing process. Otherwise, select a seed $s = G(S_{\text{cur}}, n_{\text{cur}}) \in S_{\text{cur}}$, and for all $s' \in \mathbf{S}$ such that $(s, s', r, n) \in M$, mutate s into s' with probability r. If $n_{\text{cur}} < n$, terminate the fuzzing process. Otherwise, consume time $n_{\text{cur}} \leftarrow n_{\text{cur}} - n$, and add the new input to the current input set $S_{\text{cur}} \leftarrow S_{\text{cur}} \cup \{s'\}$. Repeat this step.

Let S_{final} denote the final value of S_{cur}. **We define $\textsc{Fuzzing}(G, N, S_0)$ as a random variable whose value is the set of S_{final}. We define $\textsc{Reached}(G, N, S_0)$ as a random variable whose value is the set of program properties satisfied by at least one input in S_{final}:** $\{p \mid p \in \textsc{Target},\ \exists s \in S_{\text{final}}$ **such that $\textsc{Hold}(p, s)$ is satisfied**$\}$. We assume that if $S_0 \subseteq S_0'$ then $\Pr(\textsc{Reached}(G, N, S_0)$ contains $p) \leq \Pr(\textsc{Reached}(G, N, S_0')$ contains $p)$ holds for each $p \in \mathbf{P}$.

In practical fuzzers, various strategies are employed. For example, the strategy of a coverage-guided fuzzer selects inputs that are more likely to reach previously uncovered code, while the strategy of a directed fuzzer prefers inputs that are closer to the target locations. The last assumption

Algorithm 1 Large-scale target-guided greybox fuzzing.

Input: The strategy Exploration and the time budget $N_{\text{Exploration}}$ on exploration stage , the strategy Exploitation and the time budgets $N_{\text{Exploitation}}$, β on exploitation stage, the initial input set S_0, the total time budget N_0, and the target program properties Target.

Output: The set of reached target program properties.

```
1:  procedure LTGF(Exploration, $N_{\text{Exploration}}$, Exploitation, $N_{\text{Exploitation}}$, $\beta$, $S_0$, $N_0$, Target)
2:      $stage \leftarrow$ Exploration, $S \leftarrow S_0$, $N \leftarrow N_0$
3:      while $N > 0$ do
4:          if $stage$ = Exploration then
5:              $n_{\text{stage}} \leftarrow \min(N, N_{\text{Exploration}})$, $N \leftarrow N - n_{\text{stage}}$
6:              $S' \leftarrow$ A sample from Fuzzing(Exploration, $n_{\text{stage}}$, $S$)
7:              $S \leftarrow S'$
8:              $stage \leftarrow$ Exploitation
9:          else
10:             $n_{\text{stage}} \leftarrow \min(N, N_{\text{Exploitation}})$, $N \leftarrow N - n_{\text{stage}}$
11:             repeat
12:                 $P \leftarrow$ TargetPrioritization(Target, $S$)
13:                 for $p \in P$ do
14:                     $S' \leftarrow$ A sample from Fuzzing(Exploitation($p$), $\beta$, $S$)
15:                     $S \leftarrow S'$
16:             until The time limit $n_{\text{stage}}$ is reached
17:             $stage \leftarrow$ Exploration
18:     return $\{p \mid p \in$ Target, $\exists s \in S$ such that Hold($p, s$) is safisfied$\}$
```

in our definition can be intuitively explained as follows: when using the same strategy, a larger input set makes it easier to reach more program properties. Our definition does not restrict to any specific strategy; any strategy that satisfies the above assumption can be applied. In practice, all commonly used strategies meet this criterion.

Next, we formalize large-scale target-guided greybox fuzzing (LTGF) based on the above definition. LTGF consists of two stages: the exploration stage and the exploitation stage, each employing a different strategy. We define it as follows.

Definition 3.3 (large-scale target-guided greybox fuzzing). Algorithm 1 formalizes the workflow of LTGF. The inputs to LTGF include the strategy for the exploration stage, Exploration : $\mathcal{P}(\mathbf{S}) \times \mathbb{R} \rightarrow \mathbf{S} \cup \{\perp\}$, and its time budget $N_{\text{exploration}} \in \mathbb{R}$; the strategy mapping for the exploitation stage, Exploitation : $\mathbf{P} \rightarrow (\mathcal{P}(\mathbf{S}) \times \mathbb{R} \rightarrow \mathbf{S} \cup \{\perp\})$, where Exploitation($p$) denotes the strategy for targeting program property p; two time budgets for the exploitation stage, $N_{\text{exploitation}}$, $\beta \in \mathbb{R}$; the initial input set $S_0 \subseteq \mathbf{S}$; the total time budget $N_0 \in \mathbb{R}$; and the set of target program properties Target $\subseteq \mathbf{P}$. The output of LTGF is the set of target program properties that are finally reached.

LTGF alternates between the two stages. It starts with the exploration stage, initializing the current input set S and the remaining time N (Line 2). For the exploration stage, LTGF calculates the duration of this stage n_{stage} (Line 5) and performs one fuzzing iteration using the Exploration strategy on the current input set S (Line 6). For the exploitation stage, LTGF also calculates the stage duration n_{stage} (Line 10), and within this period, it invokes TargetPrioritization(Target, S) based on the target set and the existing inputs to generate a critical subset of targets $P \subseteq$ Target (Line 12). For each target $p \in P$, it performs a fuzzing run with a time limit β using the Exploitation(p)

strategy and the current input set S (Line 15). Finally, LTGF returns all target program properties that can be reached by the generated inputs (Line 18).

The goal of the exploration stage is to cover as much code as possible, while the goal of the exploitation stage is to leverage the inputs accumulated during the exploration stage to trigger as many vulnerabilities as possible. In practical fuzzers, for example in FishFuzz [50], the Exploration strategy preferentially selects inputs that are closest to uncovered functions associated with existing targets, and the $\textsc{Exploitation}(p)$ strategy selects seeds that pass through the location of target p and have the fastest execution speed. The reason for selecting multiple seeds in each round via $\textsc{TargetPrioritization}(\textsc{Target}, S)$ is that, in real-world fuzzing, a single seed may cover multiple target locations; in this case, it can be regarded as simultaneously applying the $\textsc{Exploitation}(p)$ strategy to multiple targets during fuzzing.

The problem this paper aims to address is how to design an appropriate TargetPrioritization algorithm so as to maximize the number of program properties ultimately reached, i.e., maximize $|\text{LTGF}(\textsc{Exploration}, N_{\text{Exploration}}, \textsc{Exploitation}, N_{\text{Exploitation}}, \beta, S_0, N_0, \textsc{Target})|$. Our core idea is, in the exploitation stage, the targets p selected each time should have a high probability of being reached by the fuzzer, that is, to make $\textsc{Reached}(\textsc{Exploitation}(p), \beta, S)$ contain p.

The probability distribution of $\textsc{Reached}(\textsc{Exploitation}(p), \beta, S)$ is difficult to compute directly. Instead of attempting to solve it explicitly, we consider, for a fixed input set S, which program properties p satisfy that $\Pr(\textsc{Reached}(\textsc{Exploitation}(p), \beta, S) \text{ contains } p)$ is no less than a threshold set by us.

Definition 3.4 (A-reachable set). Given a threshold $A \in [0, 1]$, we define $\textsc{Reachable}_A(S) = \{p \mid p \in \mathbf{P}, \Pr(\textsc{Reached}(\textsc{Exploitation}(p), \beta, S) \text{ contains } p) \geq A\}$.

Finally, we define a smaller subproblem, the solution to which can effectively facilitate solving the original problem: given a set of program properties (typically potential vulnerabilities in the program), we seek to predict whether these properties can be triggered with high probabilities by a fuzzer using a specific strategy within a limited time. We formally define this problem as follows.

Definition 3.5 (reachable fuzzing targets problem). Given a set of program properties $\textsc{Target} \subseteq \mathbf{P}$ and an input set S, for each program property $p \in \textsc{Target}$, predict $\Pr(\textsc{Reachable}_A(S) \text{ contains } p)$.

Our idea is to select program properties p with a high $\Pr(\textsc{Reachable}_A(S) \text{ contains } p)$ in TargetPrioritization, thereby increasing the number of target properties ultimately discovered. We will show how to use Bayesian program analysis to address this problem. Next, we first define Datalog-based program analysis, and then, based on it, further define Bayesian program analysis for solving this problem.

3.2 Datalog-Based Program Analysis

We first formalize the syntax and semantics of a Datalog program, and then describe its correspondence with the Datalog-based program analysis.

Definition 3.6 (Datalog syntax). We present the auxiliary definitions and notations of Datalog in Figure 8. A relation symbol r represents a relation type. A literal p is an n-ary atom with relation symbol r and n arguments, each being a variable $v \in \mathbf{V}$ or a constant $d \in \mathbf{D}$. A tuple t is an n-ary atom where all elements are constants. A clause c is a derivation rule with literals $l_0, l_1, \ldots, l_n$, stating that l_0 follows if $l_1, \ldots, l_n$ hold. A Datalog program $D = (I, O, R)$ includes input relations $I \subseteq \mathbf{L}$, output relations $O \subseteq \mathbf{L}$, and derivation rules $R \subseteq \mathbf{C}$.

Definition 3.7 (Datalog semantics). We present the semantics of Datalog in Figure 9. Given $D = (I, O, R)$ and tuple set T, $F_R(T)$ denotes one round of derivation using rules R. For each rule

$$\begin{aligned}
(\textit{variables}) \quad & \mathbf{V} = \{v, v_1, v_2, s, \dots\} \\
(\textit{constants}) \quad & \mathbf{D} = \{\texttt{is\_p}, \texttt{xrpnt}, \text{Line 16}, \text{Line 22}, 0, 1, \dots\} \\
(\textit{relations}) \quad & \mathbf{R} = \{\mathsf{Input}, \mathsf{Flow}, \mathsf{Memory}, \mathsf{Taint}, \mathsf{Alarm}, \dots\} \\
(\textit{literals}) \quad & \mathbf{L} = \mathbf{R} \times (\mathbf{D} \cup \mathbf{V})^* = \{\mathsf{Taint}(v), \mathsf{Flow}(v_1, v_2), \dots\} \\
(\textit{tuples}) \quad & \mathbf{T} = \mathbf{R} \times \mathbf{D}^* = \{\mathsf{Taint}(\texttt{m}), \mathsf{Flow}(\texttt{m}, \texttt{xrpnt}), \dots\} \\
(\textit{clauses}) \quad & \mathbf{C} = \mathbf{L} \times \mathbf{L}^* = \{[l_0 \,\text{:-}\, l_1, \dots, l_n], \dots\} \\
& \quad = \{[\mathsf{Taint}(v_2) \,\text{:-}\, \mathsf{Taint}(v_1), \mathsf{Flow}(v_1, v_2)], \dots\}
\end{aligned}$$

Fig. 8. Auxiliary definitions and notations of Datalog.

$$\begin{aligned}
F_R, f_c &\in \mathscr{P}(\mathbf{T}) \to \mathscr{P}(\mathbf{T}) \\
F_R(T) &= T \cup \{f_c(T) \mid c \in R\} \\
f_c(T) &= f_{[l_0 \text{:-} l_1, \dots, l_n]}(T) \\
&= \{\sigma(l_0) \mid \sigma(l_i) \in T \text{ for } 1 \le i \le n, \sigma \in \Sigma\} \\
[[D, T_0]] &= \mathsf{lfp}(F_R, T_0)
\end{aligned}$$

Fig. 9. Semantics of Datalog.

$c \in R$, $f_c(T)$ is the set of new tuples derived from T using c. Thus, $F_R(T)$ is T combined with all newly derived tuples. A *substitution function* $\sigma \in \Sigma = \mathbf{V} \to \mathbf{D}$ maps variables to constants. We extend this notation to literals by applying the substitution σ element-wise, replacing each variable with its corresponding constant and thereby converting the literal into a tuple. For $c = [l_0 \text{ :- } l_1, \dots, l_n] \in R$, if $\sigma(l_1), \dots, \sigma(l_n) \in T$, then $\sigma(l_0)$ is derivable. $f_c(T)$ collects all such derived tuples. Given input $T_0 \subseteq \mathbf{T}$, the output of Datalog program is $[[D, T_0]]$. $[[D, T_0]]$ is the least fixed point of F_R, computed by iterating $T \leftarrow F_R(T)$ from T_0 until convergence.

A Datalog-based program analysis is also a Datalog program, with the additional constraint that each tuple in $[[D, T_0]]$ corresponds to a program property. For example, Alarm(Line 16) indicates that there may be a memory error at Line 16.

Definition 3.8 (Datalog-based program analysis). A Datalog-based program analysis $\mathcal{D} = (D, \textsc{Property})$ consists of a Datalog program D and a mapping function $\textsc{Property} : [[D, T_0]] \to \mathbf{P}$. For each tuple $t \in [[D, T_0]]$, $\textsc{Property}(t) \in \mathbf{P}$ denotes the corresponding program property. We further require that for any target program property $p \in \textsc{Target}$, there exists an injective mapping $\textsc{TargetTuple} : \mathbf{P} \to [[D, T_0]]$ from target properties to tuples. For convenience, we use $[[\mathcal{D}, T_0]]$ to refer to $[[D, T_0]]$.

3.3 Bayesian Program Analysis for Reachable Fuzzing Targets Problem

We first formalize how to transform the results of a Datalog-based program analysis into a derivation graph, which will later serve as the structure of the Bayesian network. Then, we formalize the Bayesian network for predicting reachable fuzzing targets and describe how to incorporate fuzzer feedback as posterior information for the Bayesian network.

Definition 3.9 (derivation graph). Given a Datalog-based program analysis $\mathcal{D}$ and input tuples T_0, a *rule instance* is a derivation using rule $c = [l_0 \text{ :- } l_1, \dots, l_n]$. Formally, a rule instance is $([l_0 \text{ :- } l_1, \dots, l_n], t_0, \dots, t_n) \in \mathbf{C} \times \mathbf{T}^*$ where each $t_i \in [[\mathcal{D}, T_0]]$ and $\sigma(l_i) = t_i$ for some substitution function $\sigma \in \Sigma$. All rule instances form the set $\textsc{Instance}(\mathcal{D}, T_0)$. The derivation graph is $\textsc{Graph}(\mathcal{D}, T_0) = (V, E)$. Vertices V include all tuples and rule instances: $V = [[\mathcal{D}, T_0]] \cup \textsc{Instance}(\mathcal{D}, T_0)$. Edges E represent rule logic: each rule instance $i = (c, t_0, \ldots, t_n)$ has $n + 1$ edges $(t_1, i), \ldots, (t_n, i)$ and (i, t_0). Cycles may appear in the derivation graph, which significantly reduces the efficiency of probabilistic inference. Following prior work [34], we remove cycles and treat the graph as a directed acyclic graph (DAG).

The derivation graph represents the process by which Datalog-based program analysis over-approximates program semantics. Each tuple $t \in [[\mathcal{D}, T_0]]$ corresponds to a program property obtained through this semantic over-approximation. Our core idea is to associate a Bernoulli variable with each tuple t, indicating whether the corresponding program property $\textsc{Property}(t)$ can be reached by the fuzzer with high probability (i.e., whether $\textsc{Reachable}_A(S)$ contains $\textsc{Property}(t)$). We use the derivation relationships in the graph to set the conditional probabilities for each variable, thereby transforming the entire derivation graph into a Bayesian network. Each rule instance $i = (c, t_0, \ldots, t_n) \in \textsc{Instance}(\mathcal{D}, T_0)$ in the derivation graph is similarly converted into a Bernoulli variable. This variable represents whether the resulting program property $\textsc{Property}(t_0)$ can be reached by the fuzzer with high probability, given that all the premise program properties $\textsc{Property}(t_1), \ldots, \textsc{Property}(t_n)$ can also be reached by the fuzzer with high probability. In other words, it indicates whether $\textsc{Reachable}_A(S)$ contains $\textsc{Property}(t_0)$, conditioned on $\textsc{Reachable}_A(S)$ containing $\textsc{Property}(t_1), \ldots, \textsc{Property}(t_n)$. For each target program property $p \in \textsc{Target}$, $\textsc{TargetTuple}(p)$ is a tuple vertex in the derivation graph. Therefore, we can use the Bayesian network to systematically and semantically predict $\Pr(\textsc{Reachable}_A(S)$ contains $\textsc{Property}(\textsc{TargetTuple}(p)))$. We formally define the Bayesian network as follows.

Definition 3.10 (Bayesian network for predicting reachable fuzzing targets). We convert $\textsc{Graph}(\mathcal{D}, T_0)$ into a Bayesian network [17] $\textsc{Bayesian}(\mathcal{D}, T_0) = (X, Y)$. $X = \{x_v \mid v \in V\}$ is a set of Bernoulli variables. If $v = t \in [[\mathcal{D}, T_0]]$, then x_v indicates whether $\textsc{Reachable}_A(S)$ contains $\textsc{Property}(t)$. Otherwise, $v = i = (c, t_0, \ldots, t_n) \in \textsc{Instance}(\mathcal{D}, T_0)$, then x_v indicates whether $\textsc{Reachable}_A(S)$ contains $\textsc{Property}(t_0)$, conditioned on $\textsc{Reachable}_A(S)$ containing $\textsc{Property}(t_1), \ldots, \textsc{Property}(t_n)$. Let $\textsc{Prob}(i) : \mathbf{C} \times \mathbf{T}^* \to [0, 1]$ be the prior probability that a rule instance i is correct. We define the conditional probabilities for the Bernoulli variables. For rule instance $i = (c, t_0, \ldots, t_n)$, x_i is true with probability $\textsc{Prob}(i)$ if all $t_1, \ldots, t_n$ are true; otherwise, x_i is false. Let $i_1, \ldots, i_m$ be rule instances that derive tuple t. Then x_t is true if and only if at least one i_j is correct. Formally:

$$\begin{aligned}
&\Pr(x_i \mid x_{t_1} \wedge x_{t_2} \wedge \cdots \wedge x_{t_n}) = \textsc{Prob}(i)\\
&\Pr(x_i \mid \neg x_{t_1} \vee \neg x_{t_2} \vee \cdots \vee \neg x_{t_n}) = 0\\
&\Pr(x_t \mid x_{i_1} \vee x_{i_2} \vee \cdots \vee x_{i_m}) = 1\\
&\Pr(x_t \mid \neg x_{i_1} \wedge \neg x_{i_2} \wedge \cdots \wedge \neg x_{i_m}) = 0
\end{aligned}$$

Although the structure of our defined Bayesian network is the same as in previous work [4, 11, 15, 20, 47] on Bayesian program analysis, we fundamentally redefine its semantics, which distinguishes our approach from all prior work. We use the Bayesian network to predict whether each program property can be reached by the fuzzer with high probability, whereas previous work used the Bayesian network to predict whether each program property can be reached by any input. Specifically, in the Bayesian network of previous work, for each $v \in V$, if $v = t \in [[\mathcal{D}, T_0]]$, then x_v indicates whether there exists an input $s \in \mathbf{S}$ such that $\textsc{Hold}(\textsc{Property}(t), s)$ is satisfied. Otherwise, $v = i = (c, t_0, \ldots, t_n) \in \textsc{Instance}(\mathcal{D}, T_0)$, then x_v indicates whether there exists an input $s \in \mathbf{S}$ satisfying $\textsc{Hold}(\textsc{Property}(t_0), s)$, conditioned on the existence of inputs $s_1, \ldots, s_n \in \mathbf{S}$ such that $\textsc{Hold}(\textsc{Property}(t_i), s_i)$ satisfies for each i.

The prior probabilities $\textsc{Prob}$ can be defined precisely in theory, but are difficult to compute in practice. To bridge the gap between theory and practice, we employ approximate methods for estimating these probabilities, such as manual annotation by experts [34] or data-driven learning from labeled programs [15]. For example, the final inputs generated by multiple fuzzing processes can serve as labels to indicate whether each tuple holds or each rule instance is correct. These methods have been shown to be effective by our empirical evaluation. Therefore, our approach is **systematic and self-consistent** in theory. At the same time, by **approximating the only hard-to-compute module in practice**, we achieve an effective and practical implementation.

Nevertheless, when faced with the complexity of real-world fuzzing problems, a Bayesian network based solely on prior knowledge may not deliver satisfactory results. To address this limitation, we propose a method for generalizing posterior information by leveraging fuzzer feedback.

Definition 3.11 (fuzzer feedback). The evidence set of the initial Bayesian network is $E = \varnothing$. For a target program property $p \in \textsc{Target}$, if we take a sample of the random variable $\textsc{Reached}(\textsc{Exploitation}(p), \beta, S)$ and it contains p, we give positive feedback to the Bayesian network by updating $E \leftarrow E \cup \{x_{\textsc{TargetTuple}(p)}\}$; otherwise, we give negative feedback by updating $E \leftarrow E \cup \{\neg x_{\textsc{TargetTuple}(p)}\}$. At any time, for a target program property $p \in \textsc{Target}$, we can use a probabilistic inference algorithm on the Bayesian network [32] to compute $\Pr\left(x_{\textsc{TargetTuple}(p)} \mid \bigwedge_{e \in E} e\right)$ under the current evidence E in order to make predictions for the reachable fuzzing targets problem. We define three interactive operations:

(1) $\textsc{Update}(B, t, e)$ updates the evidence for variable x_t in the evidence set E to e. Here, $t \in [[\mathcal{D}, T_0]]$, and e can be true evidence ($E \leftarrow E \cup \{x_t\}$), false evidence ($E \leftarrow E \cup \{\neg x_t\}$), or no evidence ($E \leftarrow E - \{x_t\} - \{\neg x_t\}$).
(2) $\textsc{Inference}(B)$ performs inference [32] on B to compute probabilities of all variables given the current evidence E.
(3) $\textsc{Query}(B, t)$ returns $\Pr(x_t \mid \bigwedge_{e \in E} e)$, the probability that x_t is true given the current evidence E. Here, $t \in [[\mathcal{D}, T_0]]$.

The fuzzer feedback can be easily obtained during the exploitations stage (Line 15). However, since $x_{\textsc{TargetTuple}(p)}$ being true is equivalent to the fuzzer reaching p with probability at least A, which is not the same as a single sample of $\textsc{Reached}(\textsc{Exploitation}(p), \beta, S)$ containing p, our definition of the posterior information feedback introduces an error rate. We compute this error rate as follows.

Theorem 3.12. *Suppose the probability of giving positive feedback is r_+, and the probability of giving negative feedback is $1 - r_+$, then the average error rate does not exceed $r_+ + 1 - A$.*

Proof. Let the Bernoulli variables W_1 denote "giving positive feedback" and W_2 denote $x_{\textsc{TargetTuple}(p)}$. Then the average error rate is

$$W_{\text{average}} = \Pr(W_1)\Pr(\neg W_2 \mid W_1) + \Pr(W_2)\Pr(\neg W_1 \mid W_2)$$

where $\Pr(W_1) = r_+$, $\Pr(\neg W_2 \mid W_1) \leq 1$, and $\Pr(W_2) \leq 1$. $\Pr(\neg W_1 \mid W_2)$ denotes the probability of still giving negative feedback (i.e., obtaining a sample without p from $\textsc{Reached}(\textsc{Exploitation}(p), \beta, S)$) when $x_{\textsc{TargetTuple}(p)}$ holds (i.e., $\Pr(\textsc{Reached}(\textsc{Exploitation}(p), \beta, S) \text{ contains } p) \geq A$). Thus, $\Pr(\neg W_1 \mid W_2) \leq 1 - A$. Therefore, we have:

$$\begin{aligned} W_{\text{average}} &= \Pr(W_1)\Pr(\neg W_2 \mid W_1) + \Pr(W_2)\Pr(\neg W_1 \mid W_2) \\ &\leq \Pr(W_1) + \Pr(\neg W_1 \mid W_2) \\ &\leq r_+ + 1 - A \end{aligned}$$

□

We can make a conservative estimate: a sound static analysis typically produces about 10% true positive reports, and a fuzzer, due to its mutation capabilities, can generally discover only around

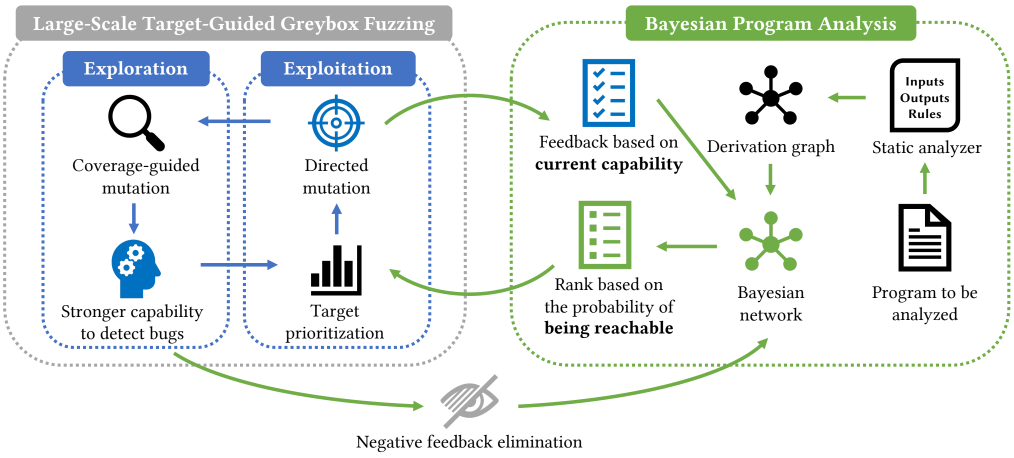
Large-Scale Target-Guided Greybox Fuzzing
Exploration
Exploitation
Coverage-guided mutation
Directed mutation
Stronger capability to detect bugs
Target prioritization
Bayesian Program Analysis
Feedback based on current capability
Derivation graph
Inputs Outputs Rules
Static analyzer
Rank based on the probability of being reachable
Bayesian network
Program to be analyzed
Negative feedback elimination

Fig. 10. Overview of framework Bayzzer for guiding large-scale target-guided grey-box fuzzing.

10% of vulnerabilities. Thus, $r_+ = 0.1 \times 0.1 = 0.01$. If we set $A = 99\%$, the average error rate is $W_{\text{average}} \leq r_+ + 1 - A = 0.01 + 1 - 0.99 = 0.02$.

In summary, we have rigorously defined a Bayesian program analysis. This analysis uses Bayesian networks to semantically predict the reachable fuzzing targets problem. At the same time, it can generalize feedback from the fuzzer as posterior information with a very low error rate. We will describe in the next section how to use this analysis to guide LTGF thus enhance its vulnerability detection capability.

4 The Bayzzer Framework

Recall that the problem this paper aims to solve is how to optimize target prioritization so that large-scale target-guided greybox fuzzing (LTGF) can ultimately reach more vulnerabilities. Figure 10 presents the workflow of Bayzzer for addressing this problem. Bayzzer predicts the probability that each target program property is reachable by the fuzzer through Bayesian program analysis, prioritizing targets with higher probabilities. Meanwhile, Bayzzer processes fuzzer feedback in the following two ways to make predictions more accurate: (1) Bayzzer generalizes the directed fuzzing results for each target during the exploitation phase as posterior information; (2) after each exploration phase, when the fuzzer has accumulated a large number of seeds and its capability has increased, previous negative feedback may become inaccurate, so Bayzzer removes them to ensure the precision of its predictions. In this section, we present the framework of Bayzzer. We begin by describing the overall workflow of our framework in Section 4.1, followed by the formalization of the target prioritization algorithm in Section 4.2. Finally, we present the details of how fuzzer feedback is processed in Section 4.3.

4.1 Overall Workflow

Algorithm 2 formalizes the workflow of Bayzzer. The workflow of Bayzzer is based on LTGF. Compared with LTGF, the inputs of Bayzzer are extended with an additional Bayesian network $B = \textsc{Bayesian}(\mathcal{D}, T_0)$, where $\mathcal{D}$ is the Datalog-based program analysis and T_0 is its input tuples, obtained via the compiler front-end or through a pre-analysis. The output of Bayzzer is the same as that of LTGF, namely, the set of target program properties ultimately reached by the fuzzer. All lines that differ from LTGF are highlighted in gray shade. First, in the exploitation stage, Bayzzer

Algorithm 2 The Bayzzer framework.

Input: The strategy Exploration and the time budget $N_{\text{Exploration}}$ on the exploration stage , the strategy Exploitation and the time budgets $N_{\text{Exploitation}}, \beta$ on the exploitation stage, the initial input set S_0, the total time budget N_0, the target program properties Target, and the Bayesian network B for predicting reachable fuzzing targets.

Output: The set of reached target program properties.

```
1:  procedure Bayzzer(Exploration, $N_{\text{Exploration}}$, Exploitation, $N_{\text{Exploitation}}$, $\beta$, $S_0$, $N_0$, Target, $B$)
2:      $stage \leftarrow$ Exploration, $S \leftarrow S_0$, $N \leftarrow N_0$
3:      while $N > 0$ do
4:          if $stage =$ Exploration then
5:              $n_{\text{stage}} \leftarrow \min(N, N_{\text{Exploration}})$, $N \leftarrow N - n_{\text{stage}}$
6:              $S' \leftarrow$ A sample from Fuzzing(Exploration, $n_{\text{stage}}$, $S$)
7:              $S \leftarrow S'$
8:              $stage \leftarrow$ Exploitation
9:              Reconstruction($B$, Target)
10:         else
11:             $n_{\text{stage}} \leftarrow \min(N, N_{\text{Exploitation}})$, $N \leftarrow N - n_{\text{stage}}$
12:             repeat
13:                 $P \leftarrow$ TargetPrioritization($B$, Target)
14:                 for $p \in P$ do
15:                     $S' \leftarrow$ A sample from Fuzzing(Exploitation($p$), $\beta$, $S$)
16:                     $S \leftarrow S'$
17:                 FuzzerFeedback($B$, $S$, $P$)
18:             until The time limit $n_{\text{stage}}$ is reached
19:             $stage \leftarrow$ Exploration
20:     return $\{p \mid p \in$ Target, $\exists s \in S$ such that Hold$(p, s)$ is safisfied$\}$
```

ranks the reachable fuzzing targets based on their predicted probabilities obtained from Bayesian program analysis (Line 13). Then, Bayzzer uses the results of directed fuzzing on these targets as feedback to the Bayesian network, improving the accuracy of predictions (Line 17). Finally, after each round of the exploration stage, the input set S increases significantly. Bayzzer updates its prediction of reachable fuzzing targets for the new input set by reconstructing the fuzzer feedback accordingly (Line 9). In Section 4.2, we describe the specific target prioritization algorithm (Line 13), and in Section 4.3, we detail how to handle fuzzer feedback (Line 9 and Line 17).

4.2 Target Prioritization

Algorithm 3 formalizes the target prioritization algorithm used in Bayzzer. Given a Bayesian network B and a target set Target, the algorithm outputs a prioritized subset of targets P. Bayzzer first performs probabilistic inference on the Bayesian network B to compute the probability distribution over relevant variables (Line 2). Then, Bayzzer maintains a critical target set P (initially empty) and a set of targets without feedback (i.e., no evidence assigned to $x_{\text{TargetTuple}(p)}$ in the Bayesian network B), denoted $P_{\text{N}} \subseteq P$ (Line 3). For each target $p \in P_{\text{N}}$, Bayzzer computes the probability r_p that the target p is reachable, based on the inference results from B (Line 5). Finally, Bayzzer selects the top-$\alpha \cdot |\text{Target}|$ targets (rounded down) with the highest probabilities into P (Line 7), where $\alpha \in (0, 1)$ is a tunable hyperparameter. If P_{N} contains fewer than $\alpha \cdot |\text{Target}|$ targets, all of them are included in P.

Algorithm 3 The target prioritization algorithm in the Bayzzer framework.

```
Input: The Bayesian network $B$ and the target set Target.
Output: The critical target set $P$.
1: procedure TargetPrioritization($B$, Target)
2:     Inference($B$)
3:     $P \leftarrow \varnothing, P_{\mathrm{N}} \leftarrow \{p \mid p \in \textsc{Target}, \text{No evidence on } x_{\textsc{TargetTuple}(p)}\}$
4:     for $p \in P_{\mathrm{N}}$ do
5:         $r_p \leftarrow \textsc{Query}(B, \textsc{TargetTuple}(p))$
6:     Let $l_i$ be the $i$-th largest target $p \in P_{\mathrm{N}}$ ranked by $r_p$
7:     for $i = 1 \rightarrow \min(|P_{\mathrm{N}}|, \lfloor \alpha \cdot |\textsc{Target}| \rfloor)$ do
8:         $P \leftarrow P \cup \{l_i\}$
9:     return $P$
```

Algorithm 4 The fuzzer feedback algorithm in the Bayzzer framework.

```
Input: The Bayesian network $B$, the current seed set $S$, and the critical target set $P$.
1: procedure FuzzerFeedback($B, S, P$)
2:     for $p \in P$ do
3:         if $\exists s \in S$ such that Hold($p, s$) is satisfied then
4:             Update($B$, TargetTuple($p$), true evidence)
5:         else
6:             Update($B$, TargetTuple($p$), false evidence)
```

For a target program property $p \in P$ and the current input set S, the probability we predict is $r_p = \Pr\left(x_{\textsc{TargetTuple}(p)} \mid \bigwedge_{e \in E} e\right)$, where each evidence e comes from a sampling of Reached(Exploitation(p'), β, S') during the current exploitation stage. Here, $p' \in$ Target is the target corresponding to this sampling, and S' denotes the input set at the time of sampling. However, $x_{\textsc{TargetTuple}(p)}$ predicts whether Reached(Exploitation(p), β, S) contains p under the current input set S. Because the goal of the exploitation stage is not to further increase coverage, the input set remains largely unchanged within a single round. We therefore assume $S \approx S'$, i.e., the two are nearly identical. As a result, the predictions produced by our Bayesian network are approximate, but the associated error is minimal and can be considered negligible.

4.3 Processing Fuzzer Feedback

Algorithm 4 formalizes the fuzzer feedback algorithm used in Bayzzer. It takes as input the Bayesian network B, the current input set S and the critical target set P used in the current round of exploitation stage. For each target $p \in P$, Bayzzer checks whether a new input has been found in this round of mutation that reaches p (Line 3). If so, this indicates that the sample on Reached(Exploitation(p), β, S) contains p. In this case, Bayzzer sends a positive feedback to the Bayesian network B (Line 4). If not, this indicates that the sample on Reached(Exploitation(p), β, S) does not contain p. In this case, Bayzzer sends a negative feedback to the Bayesian network B (Line 6).

Algorithm 5 formalizes the feedback reconstruction algorithm used in Bayzzer. During the new round of the exploration stage, the fuzzer accumulates a large number of new seeds; consequently, the current input set S will differ significantly from the input set S' during the last exploitation stage. Therefore, feedback reconstruction is necessary in order to perform more accurate computations based on the current seed set S. Since if the input set $S' \subseteq S$, then

Algorithm 5 The feedback reconstruction algorithm in the Bayzzer framework.

```
Input: The Bayesian network $B$ and the target set Target.
1: procedure Reconstruction($B$, Target)
2:     for $p \in P$ do
3:         if False evidence on $x_{\text{TargetTuple}(p)}$ then
4:             Update($B$, TargetTuple($p$), no evidence)
```

we have $\Pr(\text{Reached}(\text{Exploitation}(p), \beta, S')$ contains $p) \leq \Pr(\text{Reached}(\text{Exploitation}(p), \beta, S)$ contains $p)$ holds for each $p \in \mathbf{P}$, thus $\text{Reachable}_A(S') \subseteq \text{Reachable}_A(S)$. Based on this conclusion, our reconstruction algorithm is as follows: we retain all positive feedback, because if $\text{Reachable}_A(S')$ contains p, then $\text{Reachable}_A(S)$ will also contain p. In contrast, we remove all negative feedback (Line 4), since if $\text{Reachable}_A(S')$ does not contain p, it is still possible that $\text{Reachable}_A(S)$ contains p. Intuitively, as the input set grows, the fuzzer becomes more powerful, and previously unreachable targets may become reachable. Through reconstruction, we effectively avoid inaccurate predictions.

5 Experimental Evaluation

Our evaluation aims to answer the following questions:

RQ1. How effective is Bayzzer at finding bugs compared to other fuzzers?
RQ2. What is the performance overhead introduced by the Bayesian program analysis in Bayzzer?
RQ3. As the fuzzer accumulates more seeds, we remove all negative feedback to avoid inaccurate predictions. Is this step necessary?
RQ4. Can Bayzzer discover real-world bugs?

We first describe our experimental setup in Section 5.1. Then, we answer the four research questions in Section 5.2 to Section 5.5, respectively.

5.1 Experimental Setup

We conduct our experiments on Linux machines with 256 processors (2.25 GHz) and 256 GB of RAM. We implement Bayzzer on top of FishFuzz [50]. All LTGF logic (as described in Section 3.1) is preserved exactly as in FishFuzz. We use libDAI [31] to perform probabilistic inference on the Bayesian network.

Instance analysis. We use the Datalog analysis shown in Figure 2 as the logic core of our Bayesian program analysis. To construct the input tuples: (1) We use the SVF [42] framework to build a sparse value-flow graph (SVFG) for the program. Each edge in this graph represents a potential data flow and we convert it into a corresponding Flow input tuple. (2) We use ASan [38] to detect potential memory errors. ASan inserts runtime checks at locations that may trigger memory errors. We convert each of these alarm sites into a corresponding Alarm input tuple. (3) We conservatively assume all variables may be influenced by inputs by adding an Input(v) tuple for each variable v. This simplifies control-dependence handling, avoiding missed taints when input values affect control flow (e.g., loop conditions). While this leads to an imprecise taint analysis and preserves all ASan alarms, it remains effective since Bayesian program analysis relies more on the derivation graph structure than the analysis precision [47].

Hyperparameter. We set the selected ratio α of critical targets to 0.25. Our experiments show that this choice of hyperparameter leads to strong performance. In practice, fuzzing tools often involve many such hyperparameters. These can be tuned by evaluating different settings on similar programs to identify more effective configurations.

Baselines. We compare our approach against the following baselines: (1) Prospector [49], the current state-of-the-art LTGF technique. Prospector builds upon FishFuzz and introduces enhanced strategies for each phase of the LTGF workflow. (2) FishFuzz [50], the LTGF-based fuzzer that our approach is built upon. (3) FunFuzz [45], the current state-of-the-art CGF technique. FunFuzz uses a coarse-grained non-Bayesian static analysis to assign significance scores to functions based on the call graph, guiding the fuzzer to mutate seeds more effectively. We include FunFuzz to demonstrate that our Bayesian program analysis provides stronger guidance for fuzzing than conventional static analysis. (4) AFL++ [8], which serves as the foundational fuzzer for all three baselines and our approach.

Benchmarks. We use the same benchmark suite as Prospector, which consists of 24 real-world programs. Among them, 19 programs are from UniFuzz [21], and the remaining 5 are taken from the benchmark used by FishFuzz. During compilation, we instrument all programs with ASan [38], and treat all ASan alarms as the target set for our method as well as for the two LTGF baselines (Prospector and FishFuzz). The number of targets per program ranges from 382 (`gif2tga`) to 108,495 (`MP4Box`), with an average of 21,214 targets per program.

Metrics. We repeat the experiments for each fuzzer on each program 10 times, with a runtime of 60 hours per experiment (the same as FishFuzz, and longer than the 24 hours used by Prospector). Our total experimental runtime amounts to approximately 9.86 CPU-years. We conduct each run in a Docker container, with a specific CPU core assigned. We adopt the same bug triage scheme as Prospector, where each crash is classified into a unique bug identified by a CVE or issue number based on the ASan stack trace. This mapping scheme is identical to that used by Prospector. For each bug, we record the time-to-exposure (TTE) of the first trigger during each fuzzing process. For the 10 repeated experiments, we compute the median TTE. If a bug is not triggered in any given run, its TTE is set to $+\infty$. If no more than half of the repeated experiments trigger the bug, the median TTE is also computed to $+\infty$, indicating that the corresponding fuzzer is unable to consistently trigger the bug. We compare the effectiveness of different fuzzers by counting the number of bugs uniquely discovered by each fuzzer, where a bug is considered uniquely discovered if it is the only fuzzer with a median TTE $\neq +\infty$ for that bug.

5.2 Effectiveness

We visualize the bug sets in a Venn diagram in Figure 11 to provide an intuitive view of bug discovery overlap among fuzzers. Each number in the diagram represents the size of the intersection between the corresponding sets. Bayzzer uniquely discovers 13 bugs, whereas Prospector, FishFuzz, FunFuzz, and AFL++ only find 4, 2, 2, and 1 unique bugs, respectively. The number of unique bugs discovered by Bayzzer is not only 3.25× to 13× higher than each baseline, but also 1.4× higher than the total number of unique bugs found by the other baselines combined (9 bugs), which fully demonstrates the effectiveness of Bayzzer in finding vulnerabilities. A key observation is that, beyond the 46 basic bugs found by all fuzzers, the 13 bugs discovered uniquely by Bayzzer form the largest subset, even exceeding the 8 bugs jointly found by all three LTGF-based fuzzers (Bayzzer, Prospector, and FishFuzz). This further highlights that Bayzzer transforms LTGF into a more powerful framework through the integration of Bayesian program analysis.

We present the bug discovery curves over time in Figure 12, where the discovery time of each bug is measured using its median TTE. The performance of Bayzzer in the first 40 hours shows only limited improvement compared to the two LTGF-based fuzzers. This is because, in the initial phase of fuzzing, the Bayesian network receives limited feedback, making its probability rankings less effective in guiding the fuzzer. However, as fuzzing progresses, the Bayesian network adapts based on increasing feedback, allowing it to provide more focus guidance. As a result, Bayzzer discovers a large number of bugs in the final 20 hours, significantly outperforming all other fuzzers.

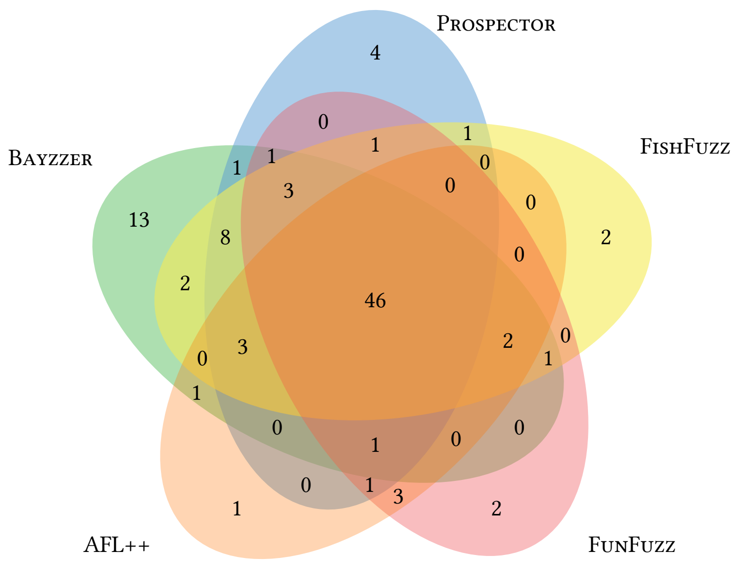

Fig. 11. Venn diagram of bugs discovered by each fuzzer.

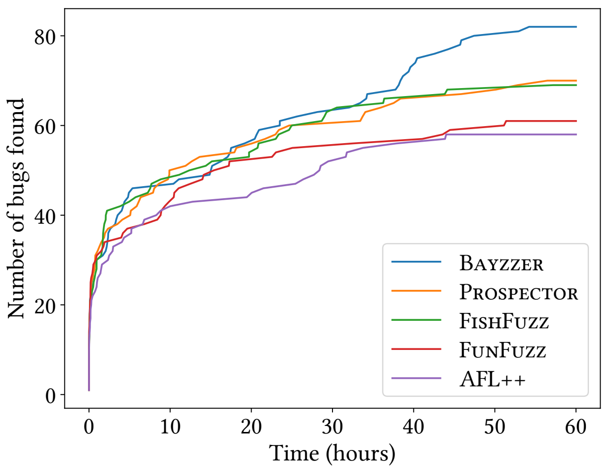

Fig. 12. Number of bugs discovered by each fuzzer throughout the entire fuzzing process. The discovery time of each bug is represented by its median TTE across 10 runs.

We present the median TTE for each fuzzer on the bugs it discovered in Table 3. Due to space constraints, we only present a subset of bugs where Bayzzer exhibits significantly better performance compared to the other baselines. We present the complete results in the appendix. We demonstrate that Bayzzer triggers bugs in multiple programs much faster than other baselines, which intuitively indicates that Bayzzer possesses a significantly different vulnerability exploration capability compared to other fuzzers. Moreover, Bayzzer achieves strong performance across various types of programs. To quantitatively support this conclusion, we follow the same evaluation strategy as used in Prospector to summarize the performance of Bayzzer against each baseline

Table 3. Median TTE (time-to-exposure) of each fuzzer over 10 runs on 24 real-world programs. N.A. (x) indicates that the median TTE is $+\infty$, and the corresponding fuzzer triggered the bug in x out of 10 runs. For each bug, the best-performing fuzzer is highlighted in bold. Due to space constraints, we only present a subset of bugs where BAYZZER exhibits significantly better performance compared to the other baselines. The complete results are provided in the appendix. In the row **Performance vs. BAYZZER**, the value a/b indicates that BAYZZER outperforms the compared fuzzer on a bugs, while the compared fuzzer performs better on b bugs. This count is used to compute the p-value from the sign test, as shown in the next row.

Program	Bug ID	BAYZZER	PROSPECTOR	FISHFUZZ	FUNFUZZ	AFL++
`jhead`	jhead-issue-8	**15m18s**	15m28s	19m18s	N.A. (2)	N.A. (1)
`wav2swf`	CVE-2017-11099	**10s**	16s	14s	17s	13s
	wav2swf-unknown-1	**1m5s**	41m52s	41m30s	1h31m	59m7s
`tiffsplit`	libtiff-issue-2243	**19m10s**	1h18m	1h44m	57m14s	2h21m
	libtiff-issue-1936	**2h21m**	18h10m	7h15m	13h59m	43h53m
	libtiff-unknown-6	**33h25m**	N.A. (4)	N.A. (5)	N.A. (5)	N.A. (4)
`MP4Box`	gpac-unknown-101	**39h23m**	N.A. (4)	57h12m	N.A. (5)	N.A. (3)
	CVE-2018-13005	**45h52m**	N.A. (1)	N.A. (3)	N.A. (1)	N.A. (0)
`nasm`	CVE-2018-8882	**20h25m**	N.A. (5)	N.A. (3)	N.A. (2)	N.A. (1)
	CVE-2018-16517	**45h44m**	N.A. (3)	N.A. (1)	N.A. (1)	N.A. (1)
`mujs`	CVE-2016-7564	**2m53s**	13m25s	4m7s	5m1s	6m58s
`tcpdump`	515bf64e	**20h57m**	N.A. (0)	N.A. (3)	N.A. (3)	N.A. (3)
	64f63920	**40h13m**	N.A. (2)	N.A. (0)	N.A. (1)	N.A. (2)
`jq`	jq-unknown-1	**53m49s**	1h32m	1h41m	1h47m	1h36m
`tic`	CVE-2017-13730	**15h11m**	21h41m	36h23m	23h2m	20h3m
`mp3gain`	CVE-2017-14407	**6m32s**	7m56s	56m7s	10m30s	11m16s
Performance vs. BAYZZER			**101**/66	**94**/70	**109**/61	**123**/43
P-value in the sign test			0.004	0.036	1.4×10^{-4}	2×10^{-10}

on a per-bug basis, as shown in the **Performance vs. BAYZZER** row in Table 3. For each bug, the fuzzers are compared based on their median TTE. If the median TTE is $+\infty$ for both fuzzers, we instead compare the number of runs in which the bug is successfully triggered. This results in a count of a/b, where a denotes the number of bugs for which BAYZZER performs better, and b indicates the opposite. We then perform a one-tailed sign test [12] using these counts, with a and b corresponding to the number of positive and negative signs, respectively. We present the resulting p-values in Table 3. All p-values are below 0.05, indicating that BAYZZER significantly outperforms the baselines in terms of bug discovery performance.

We present the median TTE for the four CVEs discussed in the two case studies in Section 2, as shown in Table 4. For the two CVEs in `mp3gain`, BAYZZER triggered both in the shortest amount of time. For the two CVEs in `tcpprep`, as described in Section 2.3, CVE-2022-27942 involves a longer call chain than CVE-2022-27941 and is therefore more difficult to trigger. BAYZZER is the only fuzzer that consistently triggered CVE-2022-27941, and also the only one that triggered CVE-2022-27942 in 4 out of 10 runs, whereas all other fuzzers triggered it at most twice. By quantitatively validating the case study observations through experiments, we further demonstrate BAYZZER's superior ability to expose hard-to-reach bugs through more effective guidance.

Table 4. Median TTE of each fuzzer over 10 runs for the four CVEs discussed in the two case studies from Section 2.

Program	Bug ID	Bayzzer	Prospector	FishFuzz	FunFuzz	AFL++
`mp3gain`	CVE-2017-14409	**4m16s**	7m4s	56m2s	10m24s	11m16s
	CVE-2017-14410	**6m13s**	8m52s	56m41s	10m40s	11m51s
`tcpprep`	CVE-2022-27941	**39h36m**	N.A. (5)	N.A. (5)	N.A. (4)	N.A. (1)
	CVE-2022-27942	**N.A. (4)**	N.A. (1)	N.A. (2)	N.A. (1)	N.A. (2)

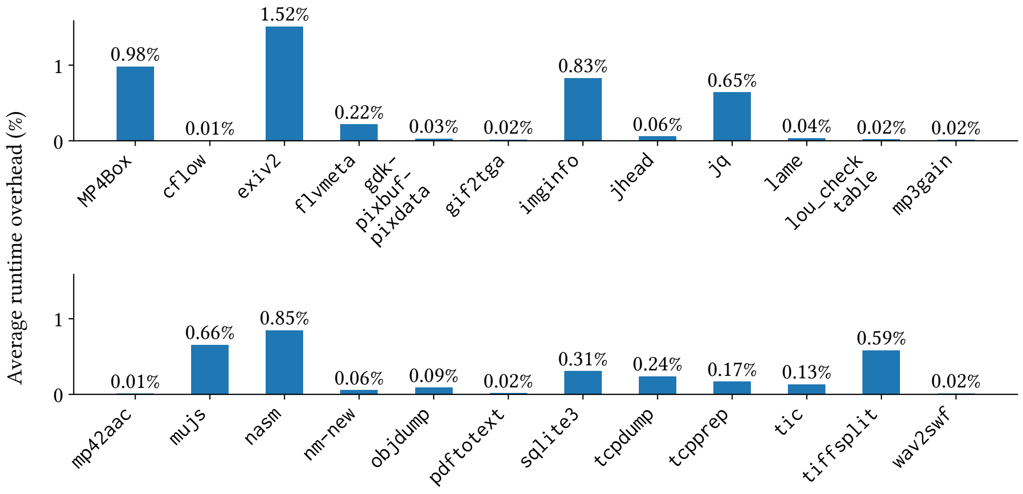

Fig. 13. The average runtime overhead introduced by Bayesian inference across 10 fuzzing runs for each program.

In summary, Bayzzer leverages Bayesian program analysis to guide LTGF, enabling it to discover bugs more effectively than other fuzzers.

5.3 Overhead of Bayesian Program Analysis

The overhead introduced by Bayesian program analysis consists of two parts: (1) the time to perform static analysis and construct the derivation graph; (2) the time spent on Bayesian inference during fuzzing. For (1), the graph construction time ranges from 0.151 seconds (`gif2tga`) to 169 seconds (`MP4Box`) across programs, with an average of 21.4 seconds. This step is performed during the compilation phase and therefore incurs no runtime cost, making it entirely acceptable. For (2), we presents the proportion of total fuzzing time (60 hours) spent on Bayesian inference for each program in Figure 13. The proportion ranges from 0.01% to 1.52%, with an average of 0.31%. The variance in inference overhead is influenced by several factors, such as the size of the derivation graph, the execution time of the program , and the stage scheduling logic in LTGF. Overall, the overhead introduced by Bayesian program analysis is negligible and fully acceptable in practice.

5.4 Necessity to Remove Negative Feedback

We conducted an ablation experiment to demonstrate the necessity of removing negative feedback. We refer to the configuration without removing negative feedback as Ablation, while all other

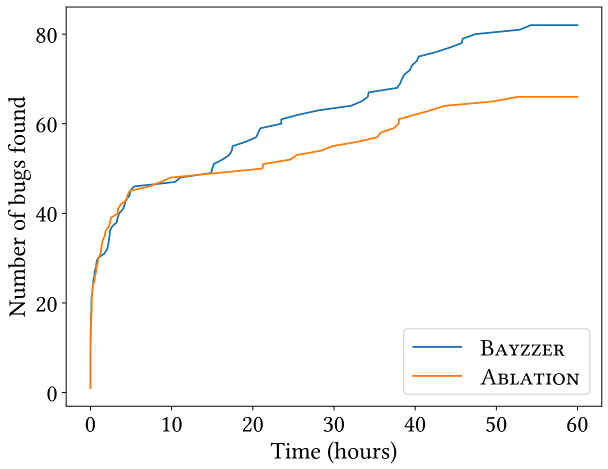

Fig. 14. Number of bugs discovered by each fuzzer throughout the entire fuzzing process in the ablation experiment. The discovery time of each bug is represented by its median TTE across 10 runs.

Table 5. New bugs found in the latest versions of the 24 real-world programs.

Program	Version	CWE	CVE
`ncurses`	6.5-20250322	Stack-based Buffer Overflow	CVE-2025-6141
`nasm`	888d9ab	Stack-based Buffer Overflow	Bug Only
`swftools`	c6a18ab	Out-of-bounds Read	CVE-2025-6271
`gdk-pixbuf`	ee5aaef0	Heap-based Buffer Overflow	CVE-2025-7345

settings are kept the same as Bayzzer. We present the bug discovery curves over time by these two fuzzers in Figure 14. During the first 15 hours, the performance of both approaches is almost indistinguishable. This is because, at this stage, the number of seeds accumulated by the fuzzer is still relatively small, resulting in limited changes in its capability, so removing negative feedback has little impact. After 15 hours, as the number of accumulated seeds becomes sufficient, removing negative feedback allows the fuzzer to refocus on reachable targets that were previously assigned negative feedback. In contrast, without removing negative feedback, these targets will not be prioritized again and thus cannot be discovered by the fuzzer. In the end, Bayzzer discovers 82 bugs, while Ablation discovers only 66 bugs. The 16 undetected bugs are the consequence of failing to remove incorrect negative feedback. Therefore, removing negative feedback can effectively improve bug discovery capability and is indeed necessary. Moreover, this experimental finding is consistent with the conclusion of recent work [37], indicating that resetting the fuzzer state can sometimes facilitate escaping from local minima.

5.5 New Bugs

We apply Bayzzer to fuzz the latest versions of the 24 real-world programs in our benchmark, as well as several popular open-source projects supported by OSS-Fuzz [10]. After one week of continuous fuzzing, we summarize the results in Table 5 and Table 6. In total, we discover 39 new bugs, all of which have been reported to the corresponding developers. The value of the vulnerabilities we

Table 6. New bugs found in popular open-source projects supported by OSS-Fuzz.

Program	Stars	Commit	CWE	CVE
`spdlog`	26.7k	3335c38	Uncontrolled Resource Consumption	CVE-2025-6140
`poco`	9.1k	530c2ef	NULL Pointer Dereference	CVE-2025-6375
`oatpp`	8.3k	c9765f9	Stack-based Buffer Overflow	CVE-2025-6566
`wasm3`	7.6k	79d412e	Out-of-bounds Write	CVE-2025-6272
`wabt`	7.4k	a60eb26	Reachable Assertion Uncontrolled Resource Consumption Use After Free	CVE-2025-6273 CVE-2025-6274 CVE-2025-6275
`draco`	6.8k	4e12ab2	Uncontrolled Resource Consumption Stack-based Buffer Overflow Out-of-bounds Read Out-of-bounds Read	Bug Only Bug Only Bug Only Bug Only
`nokogiri`	6.2k	a024cff	Heap-based Buffer Overflow Heap-based Buffer Overflow	CVE-2025-6490 CVE-2025-6494
`mruby`	5.4k	dd68681	Heap-based Buffer Overflow	CVE-2025-7207
`bloaty`	5.1k	e115514	NULL Pointer Dereference	Bug Only
`tarantool`	3.5k	46cc98b	Reachable Assertion	CVE-2025-6536
`libarchive`	3.2k	29fd918	Heap-based Buffer Overflow	CVE-2025-5915
`tidy-html5`	2.8k	d08ddc2	NULL Pointer Dereference Reachable Assertion Missing Release of Memory after Effective Lifetime	CVE-2025-6496 CVE-2025-6497 CVE-2025-6498
`plan9port`	1.7k	9da5b44	NULL Pointer Dereference	CVE-2025-7209
`libucl`	1.7k	3e7f023	Heap-based Buffer Overflow	CVE-2025-6499
`libyaml`	1k	3e7f023	Out-of-bounds Write	Bug Only
`hdf5`	760	17c16b6	Heap-based Buffer Overflow Heap-based Buffer Overflow Uncontrolled Resource Consumption Heap-based Buffer Overflow Use After Free Stack-based Buffer Overflow NULL Pointer Dereference Heap-based Buffer Overflow Missing Release of Memory after Effective Lifetime Heap-based Buffer Overflow	CVE-2025-6750 CVE-2025-6816 CVE-2025-6817 CVE-2025-6818 CVE-2025-6856 CVE-2025-6857 CVE-2025-6858 CVE-2025-7067 CVE-2025-7068 CVE-2025-7069
`librdkafka`	692	826f585	Heap-based Buffer Overflow Stack-based Buffer Overflow	Bug Only Bug Only

discovered has been recognized by the community, with 30 of them being confirmed as CVEs. We summarize Bayzzer's practical impact as follows: (1) Bayzzer has discovered a large number of vulnerabilities in popular programs, with about half of the bugs coming from GitHub repositories with more than 3k stars, and the most popular repository having up to 26.7k stars. (2) Bayzzer has found numerous high-quality vulnerabilities. For each bug, we list the associated CWE (Common Weakness Enumeration) in the table. For example, Uncontrolled Resource Consumption enables attackers to illegitimately occupy system memory, causing other processes to crash; Stack-based Buffer Overflow, Heap-based Buffer Overflow, and Out-of-bounds Write may allow illegal memory writes, potentially leading to privilege escalation or sandbox escape; Out-of-bounds Read may allow reading sensitive information from memory, resulting in information leakage. (3) Although the programs we tested have already been extensively fuzzed by the community (i.e., the 24 real-world programs) or have been continuously tested by OSS-Fuzz's 24×7 infrastructure, Bayzzer is still able to find bugs that other fuzzers cannot discover. These findings provide strong evidence of the practical effectiveness of Bayzzer in real-world software systems.

6 Discussion

In the following, we discuss potential improvements to our approach and future research directions.

Impact of static analysis precision. Our approach adopts coarse-grained taint analysis without checking for sanitization functions. This design choice reduces the computational overhead of static analysis and enables our method to be applied to a wider range of programs. Notably, the logical component of Bayesian program analysis can accommodate any static analysis based on abstract interpretation, since the derivation process can always be represented as a derivation graph and subsequently transformed into a Bayesian network. We choose Datalog as the formalism because prior work in Bayesian program analysis [34] provides a systematic method for automatically constructing a Bayesian network from a Datalog-based analysis. For analyses that do not use Datalog, additional engineering effort is required to instrument the analysis execution and export the derivations. Moreover, an interesting direction for future research is to investigate how to select an appropriate level of analysis granularity to balance prior precision, analysis efficiency, and inference efficiency, according to the specific characteristics of the program.

Handling fuzz blockers. Fuzz blockers [6, 9, 23, 35] refer to situations in which, when bug A lies on the path to triggering bug B, the program error caused by triggering bug A prevents bug B from being triggered. Our approach cannot discover such blocked bugs, so the key question is whether performance is degraded by repeatedly attempting to trigger these bugs. To address this, we calculated the proportion of runs in directed fuzzing towards high-probability reachable targets (as computed by the Bayesian program analysis) that are blocked by other unrelated bugs. Our results show that, on average, only 2.6% of runs are affected in this manner. In summary, while our approach does not attempt to solve the problem of blocking bugs, this issue does not affect its performance in practice. This finding is corroborated by our overall experimental results. A potential solution to mitigate this problem is to patch blocking bugs, as proposed by recent work [35]. Our approach is orthogonal to this method and can be combined with it to achieve even better effectiveness.

7 Related Work

Our approach is related to research on (1) Bayesian program analysis, (2) multi-target directed greybox fuzzing, and (3) techniques that leverage static analysis to enhance greybox fuzzing. We summarize the related prior works below.

Bayesian program analysis. Bayesian program analysis transforms static analysis derivations into Bayesian models and compute the probability of each alarm being true. Prior work in this area can be broadly categorized into two directions. The first line of research focuses on enabling Bayesian program analysis to generalize across diverse forms of posterior information. Eugene [26] and Bingo [34] enhance alarm ranking by learning from user feedback. Drake [11] leverages differences between code versions to improve alarm ranking. DynaBoost [4] uses dynamic execution results to refine alarm rankings. NESA [20] combines informal information with neural-symbolic reasoning to produce more accurate alarm rankings. The second line of research explores how to optimize the inference of Bayesian models for more effective alarm ranking. BayeSmith [15] applies parameter learning based on program syntactic information, while BinGraph [47] and BayesRefine [41] perform structural learning by selecting suitable abstraction. These works mainly aim to make alarm rankings more accurate so developers can inspect them more easily. In contrast, Bayzzer redefines the semantics in the Bayesian network to enable the prediction of whether each target is reachable by the fuzzer, thereby successfully extending Bayesian program analysis to guide fuzzing for fully automated bug discovery. These prior techniques are orthogonal to Bayzzer, and their advances can be used to further improve the fuzzing capabilities of Bayzzer.

Multi-target directed greybox fuzzing. Multi-target directed greybox fuzzing aims to trigger potential bugs across multiple program locations. Prior work in this area can be broadly categorized into two directions. The first direction is large-scale target-guided greybox fuzzing (LTGF), where the target set is typically large and often consists of static analysis alarms. The goal of LTGF is to discover previously unknown bugs. SAVIOR [5] integrates symbolic execution with fuzzing to mutate seeds more effectively, combining the strengths of both techniques. Bayzzer can be extended with the symbolic execution component from SAVIOR to explore paths that are difficult for fuzzers to reach. ParmeSan [33] leverages dynamic data-flow analysis to estimate the distance between inputs and multiple targets for seed prioritization. FishFuzz [50] improves precision by selecting the nearest seed for each individual target. Prospector [49] builds upon FishFuzz and introduces further optimizations across multiple phases, including target prioritization and stage scheduling. By comparison, Bayzzer introduces a principled approach to target prioritization by leveraging Bayesian program analysis, and experimental results demonstrate that it outperforms existing methods. The second direction is critical-set directed greybox fuzzing (CDGF) [1, 14, 22, 36]. CDGF typically focuses on a small set of target locations that correspond to known bugs. This setting is often used for efficiently validating or re-triggering multiple known bug within the same program, such as for regression testing or patch validation. Bayzzer can leverage techniques from CDGF by precisely guiding the selection of a smaller critical target set, enabling faster triggering of bugs.

Static analysis for greybox fuzzing. For directed greybox fuzzing (DGF), static analysis is primarily used for distance computation [2, 3, 7, 16, 18, 19, 44] and pruning of unreachable states [13, 25]. Bayzzer can be integrated with these techniques to further enhance its capability for directed bug triggering. For coverage-guided greybox fuzzing (CGF), static analysis is mainly used to guide seed prioritization [40, 43, 45], byte scheduling [24], and dictionary construction [39]. Our experimental results demonstrate that Bayzzer outperforms the state-of-the-art static-analysis-based CGF tool FunFuzz [45], providing strong evidence that Bayesian program analysis offers more effective guidance for fuzzers compared to conventional static analysis approaches.

8 Conclusion

We present Bayzzer, a framework that leverages Bayesian program analysis to guide large-scale target-guided greybox fuzzing. Bayzzer constructs a Bayesian model based on the semantics of static analysis, continuously learns from feedback during fuzzing, and computes the probability that each target is reachable by the fuzzer to prioritize target selection. We conduct experiments totaling over 9.86 CPU-years to evaluate the effectiveness of Bayzzer. Bayzzer discovered 39 previously unknown vulnerabilities in well-tested programs, 30 of which have been confirmed as CVEs. The results demonstrate that Bayzzer significantly outperforms existing fuzzers in terms of bug discovery capabilities.

Data-Availability Statement

Our artifact [48] includes all code, scripts, data, and statistics from our experiments. It contains the following:

(1) Automatic reproduction of all results from our experiments.
(2) Automated transformation of the results into Figure 11, Figure 12, Figure 13, Figure 14, as well as the complete table that contains Table 3 and Table 4.
(3) Detailed information on each newly discovered vulnerability not yet assigned a CVE, submitted to developers and presented in Table 5 and Table 6.
(4) A reusability guide for applying the Bayzzer framework to other settings and extensions.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable feedback on our paper and the anonymous artifact reviewers for their constructive suggestions. This work is supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62172017, W2411051.

References

[1] Andrew Bao, Wenjia Zhao, Yanhao Wang, Yueqiang Cheng, Stephen McCamant, and Pen-Chung Yew. 2025. From Alarms to Real Bugs: Multi-target Multi-step Directed Greybox Fuzzing for Static Analysis Result Verification. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, Lujo Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, 6977–6997. https://www.usenix.org/conference/usenixsecurity25/presentation/bao-andrew

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2329–2344. doi:10.1145/3133956.3134020

[3] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 2095–2108. doi:10.1145/3243734.3243849

[4] Tianyi Chen, Kihong Heo, and Mukund Raghothaman. 2021. Boosting static analysis accuracy with instrumented test executions. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 1154–1165. doi:10.1145/3468264.3468626

[5] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* IEEE, 1580–1596. doi:10.1109/SP40000.2020.00002

[6] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021.* IEEE, 131–142. doi:10.1109/MSR52588.2021.00026

[7] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. Windranger: A Directed Greybox Fuzzer driven by Deviation Basic Blocks. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 2440–2451. doi:10.1145/3510003.3510197

[8] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Yuval Yarom and Sarah Zennou (Eds.). USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[9] Wentao Gao, Van-Thuan Pham, Dongge Liu, Oliver Chang, Toby Murray, and Benjamin I. P. Rubinstein. 2023. Beyond the Coverage Plateau: A Comprehensive Study of Fuzz Blockers (Registered Report). In *Proceedings of the 2nd International Fuzzing Workshop, FUZZING 2023, Seattle, WA, USA, 17 July 2023*, Marcel Böhme, Yannic Noller, Baishakhi Ray, and László Szekeres (Eds.). ACM, 47–55. doi:10.1145/3605157.3605177

[10] Google. 2016. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://google.github.io/oss-fuzz/.

[11] Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. 2019. Continuously reasoning about programs using differential Bayesian inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design*

and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 561–575. doi:10.1145/3314221.3314616

[12] Myles Hollander, Douglas A Wolfe, and Eric Chicken. 2013. *Nonparametric statistical methods.* John Wiley & Sons. doi:10.1002/9781119196037

[13] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022.* IEEE, 36–50. doi:10.1109/SP46214.2022.9833751

[14] Heqing Huang, Peisen Yao, Hung-Chun Chiu, Yiyuan Guo, and Charles Zhang. 2024. Titan : Efficient Multi-target Directed Greybox Fuzzing. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024.* IEEE, 1849–1864. doi:10.1109/SP54263.2024.00059

[15] Hyunsu Kim, Mukund Raghothaman, and Kihong Heo. 2022. Learning Probabilistic Models for Static Analysis Alarms. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 1282–1293. doi:10.1145/3510003.3510098

[16] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. DAFL: Directed Grey-box Fuzzing guided by Data Dependency. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 4931–4948. https://www.usenix.org/conference/usenixsecurity23/presentation/kim-tae-eun

[17] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models - Principles and Techniques.* MIT Press. https://dl.acm.org/doi/10.5555/1795555

[18] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael D. Bailey and Rachel Greenstadt (Eds.). USENIX Association, 3559–3576. https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu

[19] Penghui Li, Wei Meng, and Chao Zhang. 2024. SDFuzz: Target States Driven Directed Fuzzing. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/li-penghui

[20] Tianchi Li and Xin Zhang. 2025. Combining Formal and Informal Information in Bayesian Program Analysis via Soft Evidences. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 143 (April 2025), 28 pages. doi:10.1145/3720508

[21] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael D. Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2777–2794. https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei

[22] Hongliang Liang, Xinglin Yu, Xianglin Cheng, Jie Liu, and Jin Li. 2024. Multiple Targets Directed Greybox Fuzzing. *IEEE Trans. Dependable Secur. Comput.* 21, 1 (2024), 325–339. doi:10.1109/TDSC.2023.3253120

[23] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. 2018. Fuzz testing in practice: Obstacles and solutions. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 562–566. doi:10.1109/SANER.2018.8330260

[24] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022.* IEEE, 1–17. doi:10.1109/SP46214.2022.9833594

[25] Changhua Luo, Wei Meng, and Penghui Li. 2023. SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023.* IEEE, 2693–2707. doi:10.1109/SP46215.2023.10179296

[26] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. 2015. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 462–473. doi:10.1145/2786805.2786851

[27] MITRE. 2017. CVE-2017-14409. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-14409.

[28] MITRE. 2017. CVE-2017-14410. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-14410.

[29] MITRE. 2022. CVE-2022-27941. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-27941.

[30] MITRE. 2022. CVE-2022-27942. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-27942.

[31] Joris M. Mooij. 2010. libDAI: A Free and Open Source C++ Library for Discrete Approximate Inference in Graphical Models. *Journal of Machine Learning Research* 11 (Aug. 2010), 2169–2173. http://www.jmlr.org/papers/volume11/mooij10a/mooij10a.pdf

[32] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. 1999. Loopy Belief Propagation for Approximate Inference: An Empirical Study. In *UAI '99: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, Stockholm,*

Sweden, July 30 - August 1, 1999, Kathryn B. Laskey and Henri Prade (Eds.). Morgan Kaufmann, 467–475. https://dl.acm.org/doi/10.5555/2073796.2073849

[33] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2289–2306. https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund

[34] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 722–735. doi:10.1145/3192366.3192417

[35] Arvind S. Raj, Wil Gibbs, Fangzhou Dong, Jayakrishna Menon Vadayath, Michael Tompkins, Steven Wirsz, Yibo Liu, Zhenghao Hu, Chang Zhu, Gokulkrishna Praveen Menon, Brendan Dolan-Gavitt, Adam Doupé, Ruoyu Wang, Yan Shoshitaishvili, and Tiffany Bao. 2024. Fuzz to the Future: Uncovering Occluded Future Vulnerabilities via Robust Fuzzing. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM, 3719–3733. doi:10.1145/3658644.3690278

[36] Huanyao Rong, Wei You, Xiaofeng Wang, and Tianhao Mao. 2024. Toward Unbiased Multiple-Target Fuzzing with Path Diversity. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/rong

[37] Nico Schiller, Xinyi Xu, Lukas Bernhard, Nils Bars, Moritz Schloegel, and Thorsten Holz. 2025. Novelty Not Found: Exploring Input Shadowing in Fuzzing through Adaptive Fuzzer Restarts. *ACM Trans. Softw. Eng. Methodol.* 34, 3 (2025), 85:1–85:32. doi:10.1145/3712186

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[39] Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. 2017. Static Program Analysis as a Fuzzing Aid. In *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10453)*, Marc Dacier, Michael D. Bailey, Michalis Polychronakis, and Manos Antonakakis (Eds.). Springer, 26–47. doi:10.1007/978-3-319-66332-6_2

[40] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2194–2211. doi:10.1109/SP46214.2022.9833761

[41] Yuanfeng Shi, Yifan Zhang, and Xin Zhang. 2025. On Abstraction Refinement for Bayesian Program Analysis. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 3232–3258. doi:10.1145/3763166

[42] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 265–266. doi:10.1145/2892208.2892235

[43] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 999–1010. doi:10.1145/3377811.3380386

[44] Valentin Wüstholz and Maria Christakis. 2020. Targeted greybox fuzzing with static lookahead analysis. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 789–800. doi:10.1145/3377811.3380388

[45] Mingxi Ye, Yuhong Nan, Hong-Ning Dai, Shuo Yang, Xiapu Luo, and Zibin Zheng. 2024. FunFuzz: A Function-Oriented Fuzzer for Smart Contract Vulnerability Detection with High Effectiveness and Efficiency. *ACM Trans. Softw. Eng. Methodol.* 33, 7 (2024), 191:1–191:20. doi:10.1145/3674725

[46] Michał Zalewski. 2013. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[47] Yifan Zhang, Yuanfeng Shi, and Xin Zhang. 2024. Learning Abstraction Selection for Bayesian Program Analysis. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 954–982. doi:10.1145/3649845

[48] Yifan Zhang and Xin Zhang. 2025. *Fuzzing Guided by Bayesian Program Analysis (Paper Artifact)*. doi:10.5281/zenodo.17784906

[49] Zhijie Zhang, Liwei Chen, Haolai Wei, Gang Shi, and Dan Meng. 2024. Prospector: Boosting Directed Greybox Fuzzing for Large-Scale Target Sets with Iterative Prioritization. In *Proceedings of the 33rd ACM SIGSOFT International*

Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024, Maria Christakis and Michael Pradel (Eds.). ACM, 1351–1363. doi:10.1145/3650212.3680365

[50] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2023. FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 1343–1360. https://www.usenix.org/conference/usenixsecurity23/presentation/zheng

Received 2025-07-10; accepted 2025-11-06