TANGO: Extracting Higher-Order Feedback through State Inference

Ahmad Hazimeh*
ahmad.hazimeh@epfl.ch
EPFL; BugScale

Duo Xu
duo.xu@epfl.ch
EPFL

Qiang Liu†
qiang.liu@epfl.ch
EPFL

Yan Wang
wangyan240@huawei.com
Huawei

Mathias Payer
mathias.payer@nebelwelt.net
EPFL

Abstract

Fuzzing is the de facto standard for automated testing. However, while coverage-guided fuzzing excels at code discovery, its effectiveness falters when applied to complex systems. One such class entails persistent targets whose behavior depends on the state of the system, where code coverage alone is insufficient for comprehensive testing. It is difficult for a fuzzer to optimize for state discovery when the feedback does not correlate with the objective.

We introduce TANGO, an extensible framework for state-aware fuzzing. Our design incorporates "state" as a first-class citizen in all operations, enabling TANGO to fuzz complex targets that otherwise remain out-of-scope. We present state inference, a cross-validation technique that distills portable coverage metrics to reveal hidden path dependencies in the target. This in turn allows us to aggregate feedback from different paths while maintaining state-specific operation. We leverage TANGO to fuzz stateful targets covering network servers, language parsers, and video games, demonstrating the flexibility of our framework in exploring complex systems. Using state inference, we shrink the scheduling queue of a fuzzer by around seven times by identifying functionally equivalent paths. We extend current state-of-the-art fuzzers, i.e., AFL++ and Nyx-Net, with state feedback from TANGO. During our evaluation, fuzzers using our technique uncovered two new bugs in `yajl` and `dcmtk`.

CCS Concepts

• **Security and privacy → Software and application security**.

Keywords

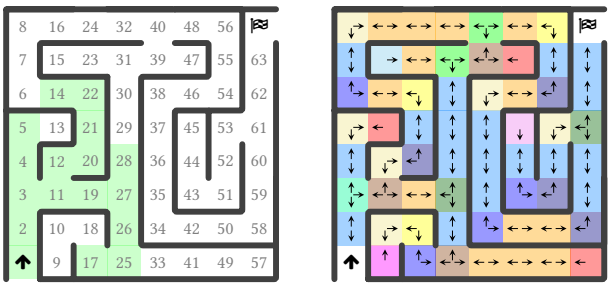
Network Protocol Fuzzing, State-aware Fuzzing, State Inference

ACM Reference Format:
Ahmad Hazimeh, Duo Xu, Qiang Liu, Yan Wang, and Mathias Payer. 2024. TANGO: Extracting Higher-Order Feedback through State Inference. In *The 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024), September 30-October 02, 2024, Padua, Italy.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3678890.3678908

*Work done prior to the author joining BugScale.
†Corresponding author.

RAID 2024, September 30-October 02, 2024, Padua, Italy

ACM ISBN 979-8-4007-0959-3/24/09
https://doi.org/10.1145/3678890.3678908



(a) W/ labeled cells, a fuzzer can systematically explore the maze. **(b) W/o labels, a fuzzer can indentify cells by their surroundings.**

Figure 1: Exploring a maze is an example of a stateful process.

1 Introduction

Fuzzing stateful systems requires special consideration. Coverage-guided fuzzing excels at finding bugs as long as feedback on covered code is strongly tied to explored functionality in the target. While this intuition holds for simple programs, the effects of consuming an input usually persist beyond the lifetime of that input in a stateful program. An observed behavior of such a stateful system is only reproducible in the context of a specific state.

We motivate stateful fuzzing with the example in Fig. 1a. If a fuzzer had access to the last reached cell as feedback, it can leverage that knowledge to prioritize paths that uncover new cells, as is the case of coverage-guided fuzzers. A stateless fuzzer would mutate a path—from the set of interesting paths it had already found—and execute it in one shot. The mutated path may or may not introduce moves along the way that would render the rest of the path uninteresting (e.g., walking into a wall). In contrast, a stateful fuzzer would select an interesting path as a prefix, follow it, then generate a move in some direction. The key difference between the two fuzzers is that the latter explores the maze incrementally, whereas the former performs a random walk, implying that the stateful fuzzer is more likely to solve the maze earlier.

Previous work reveals the benefit of state labels on fuzzer performance. IJON [3] is an annotation framework that allows fuzzers to incorporate complex state into their feedback loop. It was used to fuzz Mario Bros. in a process not too different from exploring a maze: knowledge of the last reached location guides the fuzzer towards unexplored regions. Manually annotating a target requires

effort and is often skipped in favor of readily available feedback like code coverage. While alternative techniques attempt to extract state variables from certain types of targets, varying success [5, 20, 25], code coverage remains the preferred mode of instrumentation.

However, in the absence of labels, a fuzzer may be misguided. Consider the example of an unlabeled maze in Fig. 1b. In a running session, we assume the fuzzer can request one move at a time, and its feedback is restricted to *"whether or not the player moved"*. Without knowing the label of the current cell, the fuzzer attributes the feedback only to the last generated move. For instance, if the fuzzer arrives at cell 28 through an upward move from 27, it would consider «upward» as an interesting direction and may select it more often. Yet, whether or not a player can move depends not only on the attempted move but also on its surroundings. In another iteration, if the fuzzer starts from cell 5, moving upward would yield no interesting results. Unaware of its surroundings, the fuzzer quickly exhausts the set of interesting behaviors it can observe, resulting in a random walk in exploration.

Nonetheless, we observe that such boolean feedback remains useful to model the player's local surroundings. Having found a few initial paths, the fuzzer can extract characteristics of the cells it has arrived at by trying out, at each path, all the different interesting moves it has discovered so far. A complete exploration would yield the classification represented in Fig. 1b: cells are annotated by the possible set of paths that can be followed through one move from each cell. All paths known to the fuzzer then fall into one of 14 categories based on their surroundings. In essence, the fuzzer measures the response pattern of each cell to a set of known inputs. This allows it to group its known paths by their common characteristics, e.g., paths that lead to a cell in a vertical corridor. Increasing the number of steps yields a more accurate classification such that, in the limit, each cell maps uniquely to its label. Through this process, the fuzzer extrapolates multi-dimensional feedback from a uni-dimensional metric to guide its exploration.

On the other hand, stateful fuzzers [15, 24, 26] introduce a key feature for exploring complex systems: resumability. It entails the ability to restore the target to a certain state and use that state as a starting point for further fuzzing. They achieve that through restore points, referred to as snapshots, which span different granularities, from whole-system VM snapshots, through process restore points, to record-and-replay techniques. Essentially, at each snapshot, the target occupies an implicit state as a result of the path traversed by the input. Perfect resumability ensures the reproducibility of behaviors in their respective states, and allows the fuzzer to maintain its progress while exploring different paths.

In this paper, we propose state inference to address the challenges arising from fuzzing stateful systems. State inference is a technique to produce groupings of snapshots that occupy the same implicit system state, based on similarities between input-response pairs. The key idea is to cross-test snapshots against inputs and observe their behavior to determine a mapping between snapshots and states. It does not require any prior knowledge of specification or global variables. This novel technique offers a more hands-off and effective approach to stateful fuzzing, paving the way for improved security testing of complex systems. In practice, seed queues are often biased to a subset of the functional groups discovered by the fuzzer. With the knowledge that different snapshots share the same state, the fuzzer can better model the relations between snapshots and distribute energies more equally among the inferred states. As a result, the fuzzer can schedule from the state queue first, to ensure an even exploration, and cycle faster through discovered behavior.

We implemented state inference on top of Tango, our versatile framework for stateful fuzzing. Additionally, we extended current state-of-the-art fuzzers, i.e., AFL++ and Nyx-Net, with state feedback from Tango. Our evaluation indicates that state inference significantly improves seed scheduling by effectively mapping snapshots to states, achieving an average reduction of 86.02% to 87.76% in the size of a fuzzer's scheduling queue, when fuzzing network servers and parsers. Much like solving a maze, fuzzing DOOM (see Appendix E) also benefits from the knowledge of stateful information, e.g., the player and enemy's positions, the remaining amount of ammo, and health points. By incorporating stateful feedback, Tango solves the E1M1 level in DOOM within 30 minutes and subsequently replays it in 3 minutes. We summarize our key contributions as follows:

- Introduction of the state inference, which group snapshots that belong to the same state based on their responses to inputs.
- Design of Tango, a modular-based framework for fuzzing stateful systems in a state-aware manner.
- Implementation of state-aware scheduler extensions, for AFL++ and Nyx-Net, that leverage inferred states to reduce wait times and disperse feedback, demonstrating Tango's effectiveness in analyzing complex systems.
- Open-source access to our framework and results to foster adoption and provide value to the community. Tango is available at https://github.com/HexHive/tango.

2 Challenges to Stateful Scheduling

The path explosion in fuzzing quickly degrades the performance of a fuzzer. The performance becomes even worse when we are fuzzing a stateful target due to the dependence on the system's state. The key to tackling path explosion in stateful fuzzing lies in more efficient scheduling, which faces three core challenges.

Feedback Attribution: The fuzzer gradually develops a model of the target through collected feedback, incorporating it into its input generation for further exploration. Consider the example of an FTP server in Fig. 2. If the fuzzer initially sends a correct sequence of `USER-PASS` commands, it ends up in the *AUTHED* state, where control and data transfer commands are accepted. Now in the *AUTHED* state, the fuzzer sends a control command,

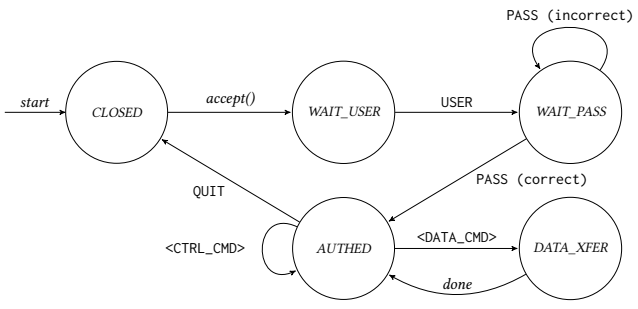


Figure 2: A simplified FTP server state diagram.

e.g. PWD, and receives positive feedback reinforcing the use of PWD in future iterations. However, the PWD command is only valid in an authenticated session context. Starting the session with any command other than USER yields a completely different behavior. Thus, without accounting for state, the fuzzer's model of the target receives conflicting or misleading feedback.

Exploration: The discovery of an interesting input can expose many new paths to the fuzzer, since that input may set up the context required to traverse those paths. Following the FTP example, if the fuzzer saves the USER-USER-PASS-PWD sequence as an interesting input, it may apply further mutations to it, leading to an input like jUnK-PASS-PWD. Unaware of the state set up by the USER command, the fuzzer mutates and destroys that part of the input, trapping itself in an error path. To avoid this loss of progress, the fuzzer should treat previous inputs as part of the state setup.

Seen differently, the history of interactions should be considered as one way to restore the current state in the target, e.g. through record-and-replay. Fuzzing is then performed incrementally along one path of exploration. Note that, while generating invalid inputs is equally important in fuzzing, this methodology does not rule out the possibility of doing so. To test for an invalid command, it suffices to start from an empty prefix path, or alternatively, only mutate past a selected prefix. Both ensure that the state set up by the prefix is not destroyed.

Soundness: If an input triggers a crash, it may not be sufficient to generate a reproducer from that input alone, since the target may have crashed due to previously accumulated state. To guarantee the soundness (reproducibility) of crashes, the fuzzer must produce an input to build up state, which requires the fuzzer to track the millions of consumed inputs within the lifetime of the target.

These core challenges can be addressed by treating *"state" as a first-class citizen* and anchoring the fuzzer's operations around it. Introducing this new dimension to fuzzing requires careful consideration and handling in the form of snapshot management, state modeling, and state-aware behavior.

2.1 Snapshot Management

For stateful fuzzing, it suffices that the target persists between successive fuzzing iterations. This allows the target to build up state through processing the consumed inputs. Nevertheless, to tackle the aforementioned challenges with feedback attribution, exploration, and soundness, stateful fuzzers typically snapshot their progress incrementally. Through these snapshots, the fuzzer can then save and restore a previously discovered path.

When fuzzing stateful targets, it is common to maintain a tree of snapshots, where each node has successors representing other snapshots. The tree is constructed by appending new snapshots as children of the current node being fuzzed whenever new feedback, e.g., control-flow edges, are discovered. Stateful fuzzers built on this idea, such as Nyx-Net [27], AFLNet [24], NSFuzz [25], and SGFuzz [5], manage their snapshot tree differently. Nyx-Net maintains an implicit tree by backtracking input sequences and injecting snapshot commands along the path. Its snapshot tree is then encoded in the input corpus itself. In contrast, fuzzers like SGFuzz, AFLNet, and NSFuzz attempt to approximate the protocol state graph by explicitly constructing a tree based on observed changes in state labels. Nodes then represent unique values of the state variable or response code, and within each node, the system maintains a set of inputs that allow the fuzzer to restore the target to a snapshot of that state. The approximate state graph is then obtained by merging tree nodes sharing the same state labels.

2.2 State Modeling

State is a semantic identifier of the system's dynamic nature. Any event or interaction with the system may update its state and modify its behavior. To measure state, three approaches exist. First, some implementations observe global variables as explicit state identifiers. However, this does not constitute a generic abstraction over states, as there are often other contributing factors. Second, since the state of a system boils down to variable memory contents which influence its responses, such as the contents of the call stack, the heap, or function-local variables, there are attempts at isolating and capturing those variables as state feedback [5, 20, 25]. However, their approaches either under-approximate or over-approximate the state. Third, it is often easier for a developer who is familiar with the target to specify their own definition of state that fits the testing goals they are trying to achieve.

Recovering the behavioral model of a system is not straightforward. To recover states and the relations between them, AFLNet requires patching the target to augment its responses such that state identifiers are explicitly indicated through the response codes. It also requires that the fuzzer is aware of how those identifiers can be extracted from the received responses. Alternatively, SGFuzz and NSFuzz instrument global variables as state indicators, but they often require manual effort to filter out noise by adding irrelevant variables to an elaborate ignore list. They also fundamentally assume that state can be consolidated to global variables, when in fact, it can span any mechanism for managing persistent memory contents, such as the call stack, function-local variables, or heap objects. On the other end of the spectrum, Nyx-Net foregoes state identification and relies solely on its high reset rates to explore more of the input space, following the blackbox fuzzing school of thought that favors execution speed over introspection.

Precise state recovery is orthogonal to fuzzing since the implementation implicit states or diverge from the prescribed protocol. Although such divergence is interesting for fault detection, it is only discernible where the specification is available, or when a baseline is used for comparison (as is the case of differential testing [18]).

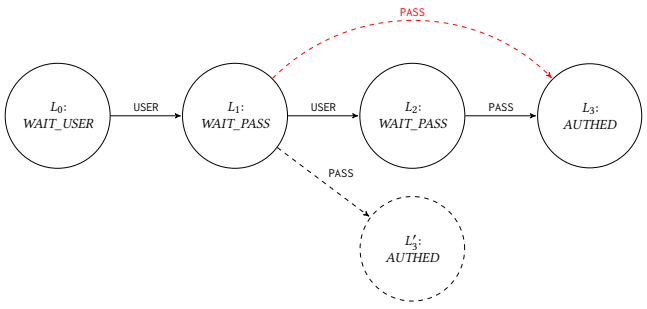


Figure 3: A snapshot tree constructed for an FTP server.

The Over-shadowed Seed: Even though there are several approaches proposed to model states, existing fuzzers are still hard to find the over-shadowed seed. Consider again the example of the FTP server in Fig. 3. When the fuzzer first sends the USER command, it takes a snapshot L_1 (in the *WAIT_PASS* state), since it observes new feedback relating to the discovery of a new command. The fuzzer then follows with USER again, taking a new snapshot L_2 due to executing the error handler in *WAIT_PASS*, and it remains in that state. Sending the PASS command then results in L_3 (in the *AUTHED* state). The fuzzer has now discovered a path to the *AUTHED* state as USER-USER-PASS. Notably, the fuzzer discovered the *WAIT_PASS* state "twice", but the *AUTHED* state only once. Depending on the type of feedback collected by the fuzzer, however, it may also be incapable of discovering the USER-PASS sequence from L_1, since the feedback for discovering the PASS command had already been attributed to L_3, and it is no longer considered an interesting signal for taking a snapshot. In effect, the discovery of the USER-PASS sequence was over-shadowed by USER-USER-PASS due to overlapping feedback. This phenomenon stacks up the snapshots and dilutes the scheduling pool.

Observation: Yet, despite L_1 and L_2 traversing different paths in the target, we know they represent the same state: the target expects a PASS in either case to transition to the next state. If we assume L_3 was never created, and we send PASS at L_1, we would discover *AUTHED* again and would create a snapshot L_3'. Conveniently, through this approach, the fuzzer discovers a shortcut to *AUTHED* requiring the minimal number of commands to reach it.

This observation is not limited to authentication routines but generalizes to any stateful system whose behavior is primarily influenced by its inputs. By probing the system at a state with different inputs and measuring its responses, we can develop a model of the state it is in. If two snapshots share the same responses across all tested inputs, we can then consider them as belonging to the same state, as far as the fuzzer is concerned.

2.3 State-aware Operation

The behavior of a stateful target changes depending on the state of the system, implying that inputs consumed by the target must be generated by accounting for the current state. Beyond using snapshots as a prefix for incremental exploration, none of the mentioned state-of-the-art fuzzers incorporates state into its operations, such as mutator schedules or state-specific dictionaries. Incorporating feedback in a state-specific manner helps develop a more accurate model of the target in each state, better guiding the fuzzer for exploring further within each state. Granted, feedback can become overwhelming for the fuzzer [9, 11, 30]. State-dependent feedback can be even more challenging to handle, as the target's behavior evolves dynamically, necessitating that feedback be incorporated dynamically as well. Nonetheless, to make the most of the collected feedback, a state-aware fuzzer should treat state as a distinct dimension for its operations across all phases of the fuzzing process.

3 State Inference

State inference models the behavior of the target through its responses to a set of inputs. Specifically, we leverage the snapshot grouping to model the target's states. At each snapshot, the target occupies a unique hidden state that drives its behavior and influences its response to inputs. For a stateful system, we posit that *two snapshots of the system occupy the same state—as far as the fuzzer is concerned—if both snapshots share the same response pattern across all tested inputs*, given a sufficient number of samples. We use this insight to develop a systematic approach for evaluating snapshots, grouping them by their observable response patterns. This grouping thus yields a hierarchy of states and snapshots that enables fairer and more targeted seed scheduling.

Definition 3.1. Feature and Feature Map: A feature is a measurable quantity that is influenced by state, e.g., edge hit count. A feature map is a mapping from a feature identifier, e.g., edge label, to its measured value.

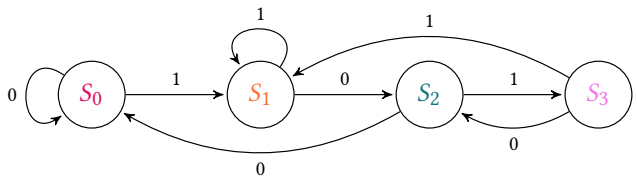
Definition 3.2. Response and Response Pattern: A *response* is a value instance of the feature map obtained by executing the target with a given input. For example, with the edge coverage map as the feature map, if three new edges are covered due to the given input, three is the response. Importantly, in our implementation, we also consider the edge's hit count as part of the response. A *response pattern* is a set of features triggered while executing an input at a snapshot. For example, the three edges newly triggered with the given input form a response pattern.

State inference has three steps. First, this grouping requires additional executions to probe the target along all the interesting paths discovered by the fuzzer. These executions are named *cross-pollination* (Section 3.1). Fortunately, however, the benefits can outweigh the overhead costs due to the nonlinearity of exercising new coverage: while fuzzing iterations grow in linear time, new features are only discovered in exponential time [6]. This means that, as the fuzzing campaign progresses, the cost of state inference is amortized over the time spent by the fuzzer between successive coverage findings. Meanwhile, the fuzzer can leverage the learned state model and the relations between snapshots for better input and mutator scheduling. Second, the grouping after cross-pollination is based on the capability matrix that stores the information obtained by cross-pollination. We come up with three operators, i.e., subsumption, elimination, and colorful collapse, to group snapshots into states (Section 3.2). Third, having obtained the static capability matrix, we have to update it during fuzzing so that the fuzzer continuously benefits from the refined feedback (Section 3.3).

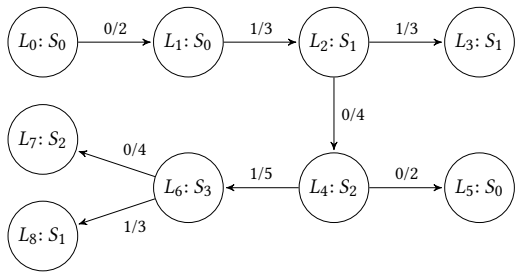
3.1 Cross-Pollination

Cross-pollination seeks hidden *capabilities* by re-applying the same input at different snapshots and observing overlaps in feedback.

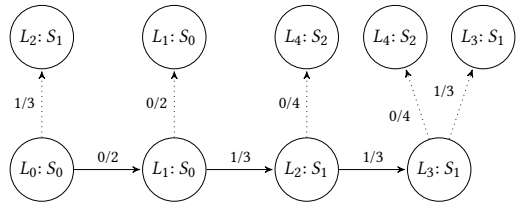
Fuzzers usually discover and record inputs that trigger new features within the target. Working under this assumption, every recorded input is guaranteed to elicit a response in at least one snapshot that was active at the time the input was first discovered. Using the battle-tested mechanism of a cumulative global feature map, where observed features are only considered interesting the first time they are encountered, the fuzzer is incapable of rediscovering the same feature across different contexts. In other words, if the same response can be observed from different snapshots, the



(a) A state diagram of a system that detects the string *b'1011*.



(b) One example of a snapshot tree constructed by the fuzzer. Each node is annotated with the implicit state L that the system occupies at that snapshot and its ground-truth state S. Each edge is annotated with the given input and the response pattern.



(c) An example of the snapshot tree after the cross-pollination. New capabilities are found, such as $L_0 \in N^-(L_2)$, $L_1 \in N^-(L_1)$, $L_2 \in N^-(L_4)$, $L_3 \in N^-(L_3)$, and $L_3 \in N^-(L_4)$.

Figure 4: A running example of cross-pollination.

fuzzer will attribute only one of those snapshots with the discovery of that response.

We illustrate this procedure with an example: consider the system with the state graph prescribed in Fig. 4a. After some simulated rounds of fuzzing, we arrive at the snapshot tree depicted in Fig. 4b. Whereas the fuzzer observed unique features—namely, state transitions in the system's finite state machine (FSM) — that resulted in this snapshot tree, we note that multiple snapshots occupy the same state. Unaware of this overlap, the fuzzer would schedule each of these snapshots individually and would integrate feedback into state-agnostic models. This results in duplicate efforts, as the fuzzer wastes many cycles on testing the same states along different paths.

However, given this initial snapshot tree, the fuzzer can leverage cross-pollination to discover hidden relations between snapshots, allowing it to better model their overlap and optimize its exploration. Specifically, given enough initial input samples, the behavior of a snapshot can be modeled by measuring its different response patterns. To discover the capabilities of all snapshots, we iterate over all inputs in the snapshot tree and apply them at every snapshot, measuring its response pattern, and recording its ability to reproduce the original response. In constructing the snapshot tree, every response pattern is associated with an edge between a parent and a child snapshot. As such, if a snapshot is capable of a response, it is equivalent to having an edge between that snapshot and the corresponding child node in the snapshot tree. Fig. 4c shows the part of the snapshot tree after cross-pollination applying 0 or 1 to all existing snapshots.

Definition 3.3. Capability: A snapshot L_i has a *capability* for L_j when the response to an input measured in L_i matches the response of another snapshot in the in-neighborhood of L_j, denoted as $L_i \in N^-(L_j)$. For example, with a snapshot tree in Fig. 4c, $L_0 \in N^-(L_2)$ holds when re-applying the same input 1 to L_0 generates the same response 3 as of the input 1 to L_2's in-neighborhood L_1. [1]

3.2 Snapshot Grouping

To continue grouping snapshots, we record all the capabilities in a *capability matrix*.

Definition 3.4. Capability Matrix A *capability matrix* is a matrix of snapshots' capabilities based on the responses to inputs and non-empty cell $\langle L_i, L_j \rangle$ stores the input that triggers a known response if $L_i \in N^-(L_j)$ holds. The left part of Fig. 5 shows a capability metric with multiple capabilities that have been inferred.

3.2.1 Subsumption. The populated capability matrix provides insight into which snapshots overlap in behavior, and consequently, how distinct snapshots are. This knowledge is essential for better guiding the scheduler toward exploring unique functionality without duplicating efforts and stalling progress. To find that overlap, we develop the subsumption operator over graph vertices ($\prec$). In short, a node u is subsumed by v iff v can replace u without affecting reachability, i.e., v has at least all the same edges as u. Taking Fig. 5 as an example, we observe that $\{L_0, L_1, L_5\}$, $\{L_2, L_3, L_8\}$, $\{L_4, L_7\}$, and $\{L_6\}$ form sets of snapshots with mutually overlapping responses. Each set of snapshots is then called an *equivalence state*. Notably, equivalent snapshots may traverse different paths through the system and thus cannot be pruned solely through power schedules [7], since state information is not captured by code coverage alone. But, any snapshot belonging to the same equivalent state overlaps, and thus can be safely eliminated, without loss of capabilities. Alternatively, to avoid inadvertently losing quality inputs, we choose to include strictly subsumed snapshots in the equivalence states of the subsuming nodes. This ensures that the fuzzer experiences the same reduction in state counts while maintaining access to all interesting snapshots. We present the formal definitions and discussions of subsumption operators in Appendix A.

3.2.2 Elimination. In practice, a fuzzer may also encounter snapshots whose response sets are proper subsets of others' response sets. From the fuzzer's perspective, such snapshots are less capable and thus not worth dedicating a scheduling slot for, despite having non-conforming behavior. Any response elicited in that snapshot can be reproduced in another having at least one additional capability. By identifying such states and eliminating them, the fuzzer further reduces the size of the scheduling queue and improves the dissemination of feedback.

[1] For a directed graph G=(V, E), a vertex u is an in-neighbor of a vertex v if (u, v) ∈ E and an out-neighbor if (v, u) ∈ E.

3.2.3 Colorful Collapse. After subsumption and elimination, we obtain a reduced set $\tilde{S}$ of equivalence states $\tilde{S}_0 = \{L_0, L_1, L_5\}$, $\tilde{S}_1 = \{L_2, L_3, L_8\}$, $\tilde{S}_2 = \{L_4, L_7\}$, and $\tilde{S}_3 = \{L_6\}$. Within each state lies a collection of snapshots that share the same behavior across all tested inputs. We model such behavior by the *capability set* of the state $\langle input : I, response : R \rangle$. For example, if $\langle I_{i,j}, R_{i,j} \rangle$ leads L_i to L_j, where L_j is in $\tilde{S}_1$, all the snapshots $\tilde{S}_1$ will share the capability set $\langle I_{i,j}, R_{i,j} \rangle$. Furthermore, in applying the input $I_{i,j}$ at any snapshot in $\tilde{S}_i$, if we can reproduce the response pattern $R_{i,j}$, we can then say that all snapshots in $\tilde{S}_i$ can reach L_j through $I_{i,j}$, because they all elicit the same characteristic response of L_j. For example, all snapshots $\tilde{S}_0$ can reach L_2 in $\tilde{S}_1$ implied by its capability set $\langle 1, 3 \rangle$.

This is directly observed when the capability matrix A_C is cast to an adjacency matrix of snapshots, as depicted in Fig. 5. To simplify the aggregation of snapshots, such that scheduling and feedback are performed at the *state* level, we can collapse the adjacency matrix through vertex contraction [23] over $\tilde{S}_i$, according to if any snapshot in $\tilde{S}_i$ can reach any snapshot in $\tilde{S}_j$.

Since the capability matrix specifies the response of every snapshot to a *single* input, we consider those as *first-order responses*; recall that in the maze analogy, we also limited the inference to one step away from each cell. The behavior of the snapshot after applying the first input can only be modeled by further cross-testing, along an additional dimension, to obtain capability tensors of higher-order responses. However, going beyond the first order significantly increases the overhead of state inference and may only partition the snapshot groupings even further; its added value is higher accuracy. As a matter of fact, we find that fuzzing seems to be tolerant to the imprecision introduced by the first-order responses due to the dampening effect of random sampling.

The *collapsed matrix* models the first-order relations between states. In our contrived example, this matrix recovers the original state graph presented in Fig. 4a. While certainly desirable, this was mainly driven by two factors:

Lossless features: Transitions in the snapshot tree coincide with those in the state graph; there are exactly 8 of each. This implies that the feature feedback to the fuzzer was capable of capturing these state transitions, without a loss in accuracy or precision. While it is not a coincidence that our tailored example displayed this behavior, it is unlikely that features are lossless in practice. The nature of the feedback (e.g., code or data coverage) dictates the accuracy and precision of the response patterns. In the FSM example, the fuzzer also managed to explore all transitions through different generated inputs. However, completeness is not guaranteed under fuzzing.

Smooth transitions: In constructing the snapshot tree, the fuzzer encountered only *new* features at any active snapshot, corresponding to triggering unseen transitions in the state graph. Had the fuzzer triggered multiple transitions in the target without observing new features, then the edge to the next recorded snapshot is not guaranteed to overlap a transition in the state graph. In effect, the fuzzer would have discovered a *path* through the state graph, rather than a single transition; yet, it would record it as one edge in the snapshot tree. It is similarly difficult to achieve "smoothness" in practice. In the maze analogy, we enforced smoothness by limiting the fuzzer to one move at a time.

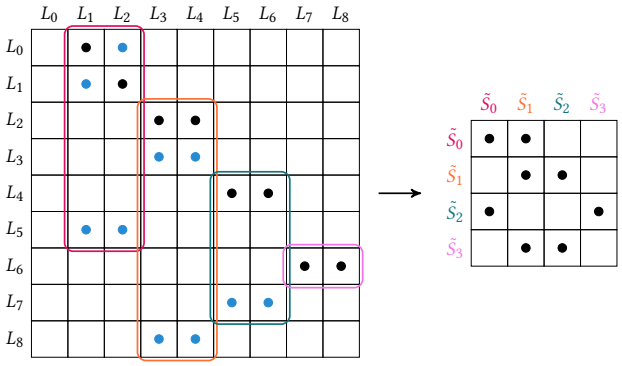


Figure 5: The capability matrix A_C (left), where ● at $\langle L_i, L_j \rangle$ indicates that $L_i \in N^-(L_j)$. It is obtained after *cross-pollination* by uncovering new capabilities—marked in blue. With *subsumption*, we group up nodes that mutually overlap in capabilities (in rectangles) in A_C. Finally, we *collapse* the matrix onto a graph of equivalence states that model the relations between snapshots (right).

To mitigate the imprecision and inaccuracy of feedback, and to avoid misguiding the fuzzer with false assumptions about state equivalence, we maintain the original snapshot tree and color it with state labels. This ensures that implicit state build-up in equivalent snapshots is not lost during collapse, and that consequent application of the state inference process does not compound the errors, but rather reduces them as more response patterns are measured and evaluated. This also allows us to iteratively collapse the same matrix to obtain a *minimal recovered model* of state relations.

3.3 Successive Rounds of Inference

After applying state inference, we obtain a capability matrix that carries the fruits of cross-pollination, along with a collapsed matrix that models the relations between labeled snapshots. This labeling, however, is static and applies only to the cross-tested snapshots. As the fuzzer progresses, it will discover more snapshots that remain unclassified and do not benefit from the results of state inference. It is then necessary that the process is continuously applied throughout the fuzzing campaign.

One straightforward approach is through batching: for every batch of m new snapshots, we re-apply the state inference, extending and updating the capability matrix from the previous round. We illustrate the state of the extended capability matrix of the second application round in Fig. 6. Note that entries in quadrant Q_A need not be revisited, which is to say, we do not recompute existing snapshot groups, since capabilities are assumed to be reproducible. After overlaying discovered edges from the latest snapshot tree onto the new capability matrix, we can continue to apply the state inference as prescribed. Alternatively, batching can be scheduled in time slots, e.g. every N minutes, accommodating for the non-linear increase in coverage by performing inference on the new snapshots generated since the last run. An adaptive hybrid approach can combine the two strategies to reduce the startup cost of inference and maximize information gained throughout a fuzzing campaign.

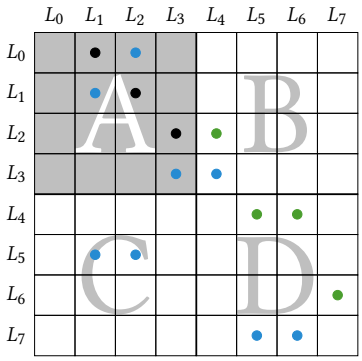


Figure 6: The capability matrix of the second round, with $m = 4$ new snapshots. The shaded quadrant Q_A contains the populated matrix from the previous round. The Q_B and Q_D quadrants are overlaid with edges—marked in green—from the latest adjacency matrix. Cross-pollinated capabilities are marked in blue. Q_C is always initially empty since old snapshots cannot have new predecessors.

While we present our approach as an independent post-discovery step, state inference can also benefit from the fuzzer itself since the fuzzer spends much time on generating inputs that do not trigger new features. Nonetheless, the feedback is often non-empty: these uninteresting inputs likely elicit known response patterns that may overlap those of other snapshots. This observation allows the fuzzer to dynamically extend the capability matrix at a minimal cost: the computational overhead of matching.

4 Overhead, Optimizations, and Trade-offs

To uncover hidden capabilities, state inference relies on cross-validation, a technique that is notorious for its quadratic complexity. Applying state inference in batches of m new snapshots requires that the fuzzer discovers m new response patterns. But coverage growth is linear in exponential time [6]: to discover m new response patterns, the fuzzer spends on the order of $\exp(m)$ more time than for the last batch. As the fuzzer continues to progress, the overhead cost of state inference is amortized over the executions performed between rounds (See Appendix B for a more formal analysis). In the meantime, it reaps the benefits of modeling state relations in scheduling and in generation. We assess these costs and benefits through our evaluation in Section 6.2 and Section 6.3.

The ramp-up cost of state inference is, however, high. A fuzzer finds the most coverage in the first few epochs of the campaign, necessitating frequent rounds of cross-testing. Whereas the overhead tapers off at the tail, the initial costs can overwhelm the fuzzer, making it spend most of its time in the beginning just on inference, and bringing its progress to a slow halt. The cost of ramping up greatly varies with initial seed coverage, the complexity of the target, and the execution speed, among other variables.

To address the ramp-up cost, we propose several optimizations that reduce the overhead of state inference, at the cost of some accuracy in grouping snapshots. The prescribed cross-testing procedure requires that all cells of Fig. 6 outside of Q_A be tested for adjacency. We propose three optimizations (one for each quadrant) to in the form of skipped tests, thereby reducing the number of resets and executions required for each round of inference.

4.1 State Broadcast (O_B)

Suppose the inferred states in Q_A are correct once constructed, then cross-testing the capabilities of individual snapshots within the same state becomes unnecessary. Working under that assumption, equivalent snapshots always have the same capabilities, which we can evaluate as a property of the state they belong to. For a state with N snapshots, it thus suffices to perform at most m cross-tests, instead of $m \times N$. Discovered capabilities in Q_B are then broadcast to all snapshots within a state, reducing the number of cross-tests (See Appendix C for a formal analysis of its reduction). In applying state broadcast as an optimization, the overhead of inference is distributed among states, rather than snapshots. This results in higher prediction accuracy for "uncommon states", i.e., those with fewer snapshots, which are arguably of more interest to the fuzzer.

4.2 Response Fingerprinting (O_C)

After one round of state inference, we obtain a capability matrix in Q_A. Each state and its associated capability set serve as labeled data for training a decision tree (DT) classifier. Such a classifier can optimally divide the input space, enabling us to primarily cross-test inputs in Q_C which maximize information gain.

We fit a DT over this training data to infer *response fingerprints*: minimal subsets of capability sets which are characteristic identifiers of states. The DT classifier yields a binary tree, where each internal node tests for a capability, such that nodes closer to the root yield higher information gain. Leaf nodes consequently carry a value indicating the most likely candidate state.

Following this procedure, for each new snapshot in the inference batch (Q_C), we traverse the DT from the root and query it for the next capability to test, until we hit a leaf node. At this point, the candidate state is identified. The tested capabilities along the path form a subset of the capability set of the candidate. Nonetheless, to reduce false positives and satisfy the rules of subsumption, we extend the testing to all non-empty responses in the candidate's capability set.

Consider the second round in Fig. 6. We fit a DT over the capability sets of $\tilde{S}_0 : \langle L_1, L_2 \rangle$ and $\tilde{S}_1 : \langle L_3 \rangle$. In this simplistic example, a test on any of $\{L_1, L_2, L_3\}$ is enough to yield a classification. With the given labels, it is thus sufficient to perform one test, instead of four, to classify a new snapshot. A snapshot that passes the test has a capability set that matches that of $\tilde{S}_1$, making it at least equivalent. However, failing the test implies an empty capability set, yet the classifier would predict $\tilde{S}_0$. To properly test for equivalence, we must test against $\tilde{S}_0$'s entire capability set in the least (for a total of three tests). This allows us to properly classify L_5 in Fig. 6. Note that, if the capability set of a new snapshot L_u matches that of the candidate state $\tilde{S}_v$ using DT classifiers, we can only infer that $\tilde{S}_v \preceq^+ L_u$, since we do not have information about what we did not test. Whether or not $\tilde{S}_v \prec L_u$, the result is then the same: a state $\tilde{S}_w = \tilde{S}_v \cup \{L_u\}$. Combined with coloring, this approach ensures that state inference remains consistent under sparsity.

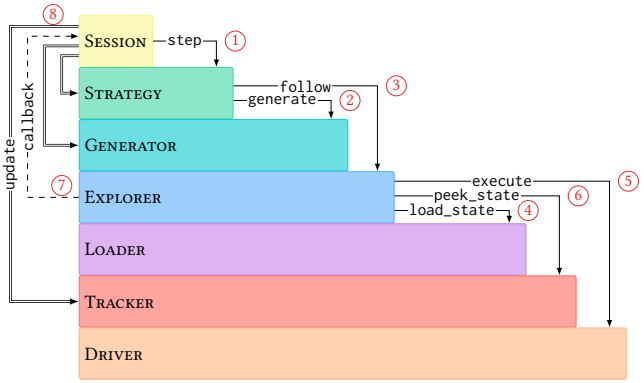


Figure 7: The general workflow of the Tango framework.

4.3 Test Extrapolation (O_D)

With response fingerprinting, we can reduce the number of tests to be done in the Q_C quadrant of Fig. 6. However, Q_D remains to be fully tested. Building on top of the DT's predictions, we can leverage Q_B to extrapolate which tests in Q_D need to be applied. Instead of cross-testing all m new snapshots against each other, we can reduce the cost of inference by extrapolating tests from the new state capabilities.

After finding a candidate state to which each snapshot belongs, we populate Q_B to explore the new capabilities of pre-labeled snapshots (i.e., those in Q_A). We follow that by testing new snapshots only against the new capabilities of their candidate states. As with response fingerprinting, this process ensures correct subsumption conditions, minimizing the risk of mislabeling snapshots.

5 Tango: The Framework

State inference extends the fuzzer's knowledge base with information about the behavior of the target, which could be leveraged to improve its exploration through (a) higher scheduling efficiency over states; (b) training of state-specific models for inputs and mutators; and (c) better approximation of state relations. Nonetheless, the technique introduces new definitions and requires components which are not explicit in existing fuzzers or frameworks, such as snapshots, states, and transitions. To assess and evaluate the feasibility and benefits of state inference, a whole re-write and restructuring of the typical fuzzing workflow is needed. The lack of an existing framework for state-aware fuzzing and the inflexibility of existing monolithic fuzzers motivated the design and development of Tango: a state-aware fuzzing framework.

5.1 Workflow

Anchored around the notion of state-sensitivity, Tango generalizes over the traditional fuzzer architecture and offers a flexible environment for developing tailor-made fuzzers of stateful systems. The workflow is presented in Fig. 7. In the context of a fuzzing Session, we (1) iteratively step through a Strategy that governs the exploration and exploitation efforts of the fuzzer. Provided with knowledge of the current state of the system, (2) the Strategy chooses a target state to fuzz and invokes the Generator to construct a candidate input, suitable for application under that state. The former then (3) forwards the input to the Explorer, which (4) ensures that the system occupies the target state, then (5) executes the input through its Driver. With the help of the Tracker, (6) peeking into the new state of the system enables the Explorer to record any observed changes. The latter then (7) forwards its findings over a callback to the Session, which (8) broadcasts any updates to concerned components, thus closing the feedback loop.

5.2 Implementation

Tango is implemented in Python 3.11 for Python's flexibility and ease-of-use. We expand on the implementation details in Appendix D. Importantly, during our implementation, we have identified three kinds of non-determinism: random numbers, time, and shared resources, and concretely addressed these issues by providing the random generator with a constant seed, providing the target with normalized time [1], and recovering shared resources when the target is reset. Removing nondeterminism that unnecessarily differentiates two snapshots that should be merged will improve the overall accuracy of the snapshot grouping.

6 Evaluation

State inference is a mechanism to identify functionally-distinct states through cross-testing its snapshots against different inputs. To assess the effectiveness of this technique and pinpoint its potential use cases, we address the following research questions through distinct evaluation campaigns:

RQ1 What is the cost of cross-testing?
RQ2 How well do fuzzers distribute cycles to functionally-distinct snapshots?
RQ3 What is the potential reduction in queue sizes?
RQ4 How does code coverage correlate to state coverage in evaluating state-aware fuzzers?

6.1 Experimental Setup

To quantify the cost and benefits of a new technique, it must be compared against a baseline where only key features are different. These features must then be tested individually. Since we implemented state inference on top of Tango, we perform the parametric analysis with Tango itself as the baseline, by toggling new features and sweeping over parameters. This enables us to measure the induced effect of each new aspect by changing one variable at a time, discounting the possible variances from runtime effects, implementation artifacts, or a richer set of mutators and schedulers, such as those in AFL++ [10].

To that end, we tackle **RQ1** through a set of experiments on Tango-Infer, configured to use state inference as a strategy. Moreover, to assess the potential inefficiencies of scheduling functionally-equivalent snapshots (**RQ2**), we dispatch a set of campaigns to state-of-the-art state-aware fuzzers, collect their seed queues, and replay them through Tango-Infer to find functional groupings among the fuzzers' snapshots and analyze the skew of snapshots in explored states. Additionally, for **RQ3**, we use the results of state inference on fuzzer queues to approximate the expected reduction ratio α encountered across different targets. Finally, to assess the practical advantage of state inference, we set up augmented fuzzing campaigns on top of state-of-the-art baselines, where state inference is run periodically to condense the seed queue. To answer

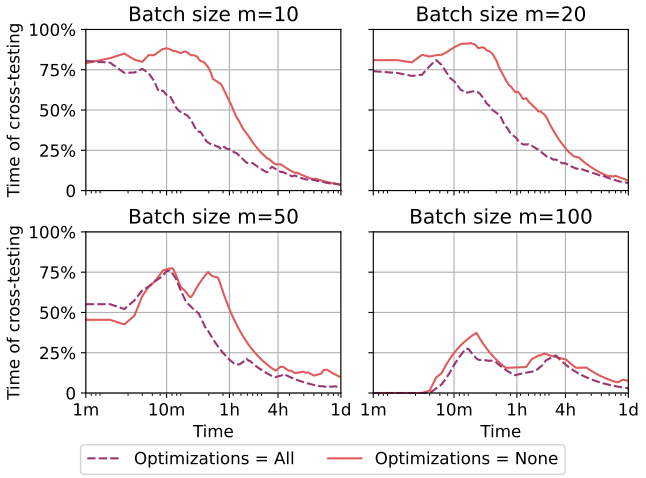
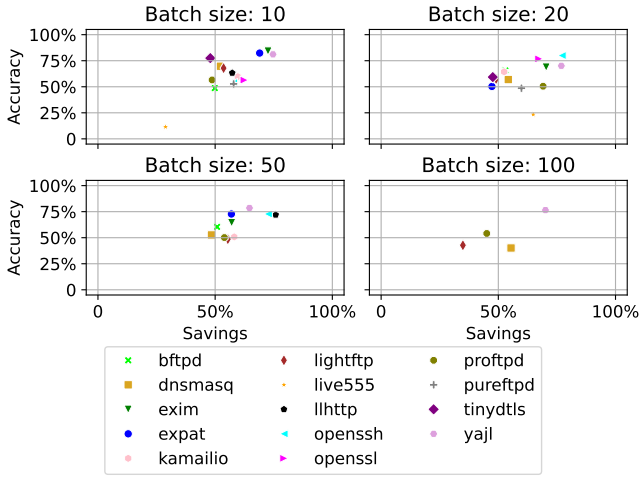
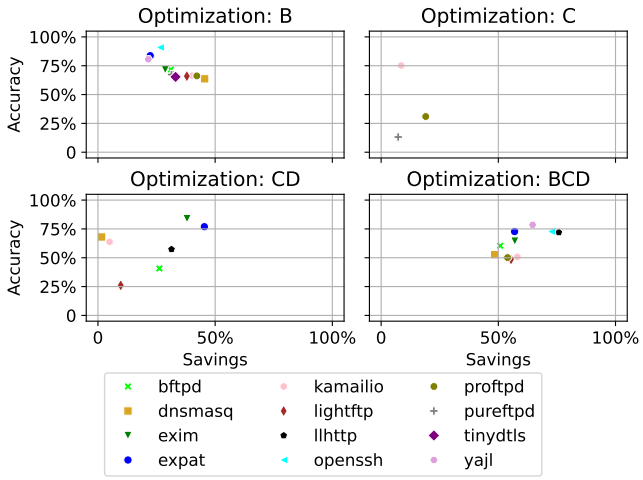


Figure 8: The median overhead of state inference as the proportion of time spent on cross-testing, when optimizations are disabled (solid lines) and enabled (dotted lines), over 24 hours, under different batch sizes. Suppose that the speed to discover new snapshots is the same, the lower m is, the more frequently the state inference is performed.



(a) Optimizations: All



(b) Batch size m=50

Figure 9: The hit accuracy (the percentage of correctly labeled snapshots) of optimizations as a function of introduced savings (the skipped tests) based the ground truth collected when doing state inference without optimization.

RQ4, we report the achieved coverage with and without inference, and we measure the benefit of state inference as the proportion of equivalence states discovered uniquely by each fuzzer.

We perform evaluations against all the thirteen version-anchored targets from ProFuzzBench [21] and three stateful parsers: `libexpat`, `yajl`, and `llhttp`. While parsers are traditionally considered stateless, our evaluation highlights their stateful nature, as well as TANGO's flexibility in fuzzing diverse data channels. We run 24-hour campaigns, each with 3 trials to account for randomness. We conduct all experiments on four servers, each with 32 Intel Xeon Gold 5218 CPU (2.30GHz) cores, 64GB RAM, and Ubuntu 22.04.

6.2 RQ1: Empirical Overhead

During inference, the target is reset, and inputs are executed for cross-testing every cell in the capability matrix. To assess the overhead of this operation, we measure the time spent by the fuzzer and the number of tests performed, across varying settings of batch size m and enabled optimizations O_B, O_C, and O_D.

Fig. 8 shows the overhead of state inference as a function of time. The beginning of a fuzzing campaign records many snapshots and frequently invokes state inference, due to the initial seeds quickly expanding coverage. As exploration speed tapers off, the fuzzer continues to generate and execute inputs, progressively spending less of its time on cross-testing. However, it leverages the knowledge gained from previous applications of state inference to schedule its exploration more evenly across the functionally distinct snapshot groups. As discussed in Section 3.3, it is essential to continuously apply inference on new snapshots. Otherwise, the fuzzer regresses in the direction of high-density regions [7]. Fig. 8 highlights optimizations can reduce the ramp-up cost and thus allow the fuzzer to resume regular operation faster.

To better understand the influence of the optimizations, Fig. 9 breaks down the trade-off between the introduced savings (skipped tests) and the sacrificed accuracy (the percentage of correctly labeled snapshots). First, we lack data for some targets since the number of the snapshots generated during fuzzing these targets is insufficient to perform cross-testing and the validation of savings and accuracy. Second, as shown in Fig. 9a, the optimizations can decrease more than half of the tests while maintaining moderate accuracy. Third, for a batch size of 50, Fig. 9b shows that various optimizations impact savings and accuracy differently.

Optimizations rely heavily on the results of previous rounds of state inference. If a grouping is incorrectly established, it could remain divergent for the lifetime of the fuzzing campaign, especially since, in our implementation, matching is not error-tolerant. If the fuzzer falsely generates distinct groupings, then future rounds of

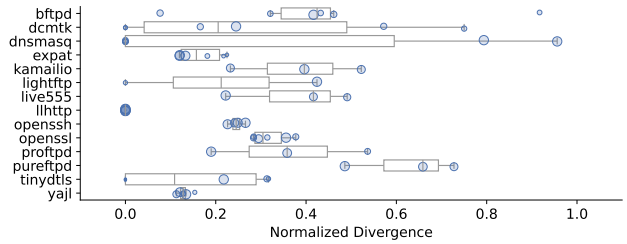


Figure 10: Normalized values of the Kullback-Leibler divergence from uniformity of observed snapshot-to-state distributions, illustrated through notched box-plots at 95% CI. Divergence is calculated individually, for each seed queue and data points are then overlaid as a ○ scatter plot. The size of each marker is proportional to the number of snapshots N, normalized by the maximum number of snapshots observed for its target across all campaigns.

inference, and optimizations within them, could propagate the errors (unless groups are merged again under subsumption).

6.3 RQ2: Snapshots in Biased Queues

Whenever an evolutionary fuzzer encounters interesting coverage, it saves the input that caused it in its seed queue. For stateful fuzzers, these seeds serve as snapshots to restore the target to the reached state. To quantify the benefits that seed scheduling through state inference could provide, we measure the distribution of fuzzer-generated snapshots across functionally-distinct equivalence states. If we assume that a state-unaware fuzzer selects seeds from the queue with a uniform distribution (i.e., it assumes that seeds are evenly spread across target functionality), then we can assess the "surprise" [28] of sampling uniformly from a skewed population. We calculate $\hat{D}_{KL}(\mathbb{S}||\mathbb{U})$, the Kullback-Leibler divergence [14], of the observed $\mathbb{S}$napshot-in-state distribution against a $\mathbb{U}$niform reference, normalized by $log(N)$, where N is the total number of snapshots observed in each campaign.

In this experiment, we run AFL++, Nyx-Net, and Tango-Infer against compatible targets, collect their seed queues, and apply state inference to extract snapshot groupings. We present the results in Fig. 10. A value $\hat{D}_{KL} = 0$ indicates that sampling the seed queue uniformly yields results consistent with sampling a uniformly distributed population, i.e., where there are equally as many snapshots for every equivalence state discovered by the fuzzer. On the other end of the spectrum, $\hat{D}_{KL} = 1$ implies that uniform sampling yields the highest surprise: whereas the fuzzer would expect to be exploring different functionalities by cycling through its queue, it is likely tunnel-visioned by a majority equivalence state. A fuzzer that samples its seed queue uniformly would be hindered by duplicate efforts and a self-reinforcing equivalence state.

On the other hand, while fuzzers generally employ more complex seed scheduling mechanisms [7, 16, 29], those cannot replace state awareness. A schedule that prioritizes snapshots unequally based on observed feedback inherently disregards the possible overlap of those snapshots with others in their equivalence state. Equivalent snapshots exercise overlapping behavior, insofar as cross-testing has not identified discrepancies that necessitate subdividing the

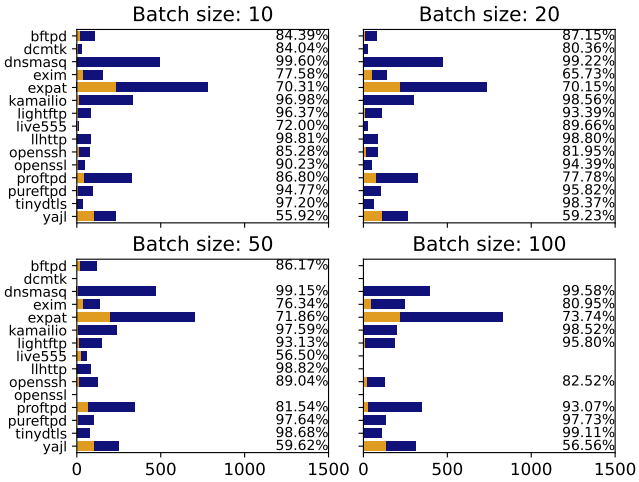


Figure 11: The number of the discovered snapshots (in blue) and the number of the inferred states (in orange). Text after each bar shows the number of the inferred states and the reduction ratio $\alpha = 1 - {}^{\text{states}}/_{\text{snapshots}}$ at the end of 24-hour fuzzing campaign without optimizations.

group into distinct functionalities. However, since fuzzing is incomplete, it may be that now-equivalent snapshots may diverge in the future, given that they are sufficiently scheduled. A non-uniform scheduling strategy may starve those snapshots of the time needed to explore their potential capabilities, often in favor of the *first* one which uncovered interesting features.

6.4 RQ3: Reduction Ratio α

The main advantage of state inference is condensing the seed queue into functionally distinct islands. This allows seed scheduling to be more balanced and reduces redundant exploration of the same code regions. To assess the effect of state inference on queue size reduction, we measure the number of snapshots generated by every campaign, and the corresponding number of discovered groupings. In Fig. 11, we report the reduction ratio α as $\alpha = 1 - {}^{\text{states}}/_{\text{snapshots}}$.

The average reduction ratio is 86.02% ($m = 10$), 86.04% ($m = 20$), 85.08% ($m = 50$), and 87.76% ($m = 100$), i.e., the queue is around seven times smaller. Combined with the results from Section 6.2, this suggests that, during later fuzzing stages, the fuzzer can cycle through its queue around seven times as fast, at a diminishing cost. The skewed distribution of seeds in fuzzing queues, suggests that state-of-the-art fuzzers could benefit from applying state inference to prune their queues and avoid tunnel vision towards high-density regions. Continuous incremental application also ensures that new snapshots are incorporated and that the fuzzer avoids regression towards non-uniformity.

6.5 RQ4: Case Study on Cross-Inference

We implement state inference without optimizations as a hotpluggable component to introduce state-aware scheduling to two existing fuzzers: AFL++ (for the streaming parsers) and Nyx-Net (for the network servers). The fuzzer and Tango share one physical core throughout the campaign. Tango continuously checks for

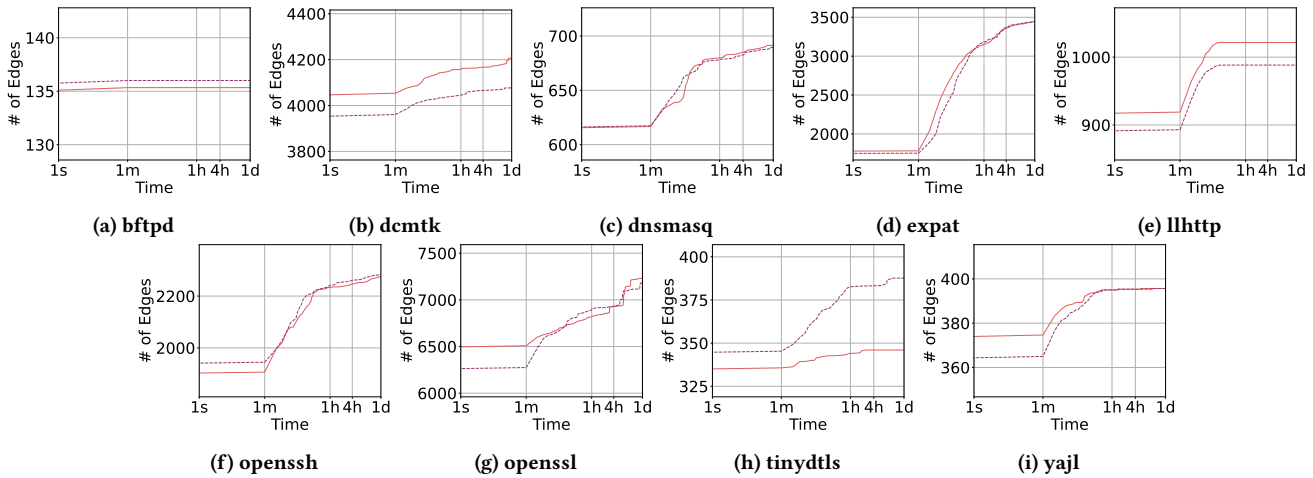


Figure 12: Edge coverage collected from Nyx-Net (for network servers) and AFL++ (for parsers) when running without (solid lines) and with (dotted lines) the state inference extension.

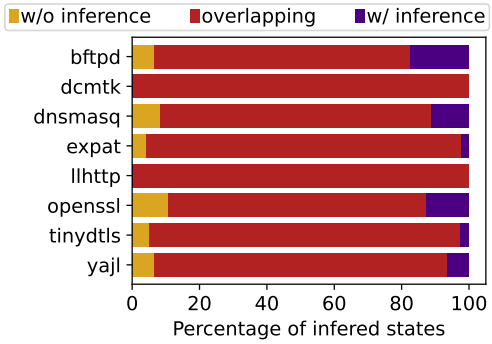


Figure 13: Cross-inference results showing the distribution of overlapping and unique behaviors discovered by stateful fuzzers with and without state inference.

new inputs in the fuzzer's seed queue and applies state inference, exporting its results for use by the fuzzer's scheduler. TANGO is also configured to export any interesting inputs it discovers during cross-testing, further reducing its effective overhead.

We measure the code coverage achieved by both the unmodified and the augmented variants of the fuzzers, and we present those in Fig. 12. Since code coverage alone cannot capture state information, we leverage state inference as a performance metric, through a process we call *cross-inference*. Seeds obtained from two competing fuzzers are used to construct a snapshot tree, upon which state inference is applied. We interpret the overlap and disjunction of those snapshots in equivalence states as a measure of state coverage: states where all snapshots of fuzzer **B** are subsumed by snapshots of fuzzer **A** are considered unique to **A**, without loss of generality. Otherwise, we consider them overlapping. The results of this experiment are illustrated in Fig. 13.

The experiments yield an interesting result: despite both fuzzers attaining similar code coverage, state inference revealed distinctions in uncovered functionality, favoring the state-guided fuzzers. The additional code covered by the unmodified fuzzers may not directly translate to state coverage: "novel" paths may belong to the same equivalence state. Through our evaluation, state-guided fuzzers uncovered two new bugs: a heap buffer overflow in `dcmtk` and a heap out-of-bounds read in `yajl`.

7 Discussion

State inference introduces a new metric for a fuzzer to optimize its progress and performance on stateful systems. Consequently, it raises the question of whether such a metric ultimately improves the fuzzer's ability to find bugs. Notably, we argue that *optimizing state coverage requires state-aware metrics*, although not exclusively.

Code vs State coverage: Our case study on cross-inference highlights a key observation: code coverage is not sufficient for stateful exploration. Despite achieving higher code coverage, state-unaware fuzzers discovered fewer behaviorally-unique states, as seen in Fig. 13. This is further justified by our takeaway from Fig. 10, that seeds in the queue of coverage-guided fuzzers are non-uniformly distributed. Stateless programs can be seen as occupying only one state, and code coverage enables in-depth exploration of that state. Stateful programs introduce a new dimension for fuzzers. So, despite state-unaware fuzzers achieving slightly higher code coverage, cross-inference results highlight that state-aware exploration better maximizes state coverage. A scheduler may incorporate other metrics like code coverage to prioritize individual snapshots or diversify exploration within a state.

Bug-finding advantage: In `dcmtk`, TANGO uniquely detected a clean-up bug, which was subsequently reported and fixed. To trigger the bug, sending any message and waiting for clean-up is insufficient; instead, it requires setting up a valid state, followed by a disconnection. TANGO's "post-mortem tracking" was also pivotal for achieving this: it monitors decommissioned targets asynchronously until they crash or exit, without impacting performance. In `yajl`, 4 of 5 TANGO-AFL++ campaigns reliably triggered the bug, whereas

it was never triggered by the unmodified AFL++. In a later evaluation with 20 campaigns, the bug was triggered in 14 Tango-AFL++ campaigns, but only in 4 campaigns w/o inference. This significant difference underscores the benefit of state-aware scheduling. Without ground-truth benchmarks, we cannot assess fuzzer performance through new bugs alone, as the total bug count is unknown. Instead, we report that fuzzers with state inference found a superset of the bugs found by fuzzers without it. Tango provides the tools necessary to perform state-aware fuzzing, and we believe it is an important foundation for future research.

Exploration vs Exploitation: State inference offers a mechanism for assigning a label to each snapshot. This enables balanced exploration among states to avoid the starvation of less frequent ones. A power schedule over states, like AFLFast [7], could then incorporate the frequency of encountering a state to prioritize the exploration of low-frequency behaviors. If state coverage is the only metric being optimized, then equivalent snapshots would be indistinguishable, and a uniform schedule within the same state would be reasonable. However, a fuzzer can still schedule equivalent snapshots non-uniformly based on code coverage and occasionally prioritize weaker ones.

Misclassification: A false grouping may occur either due to (i) insufficient cross-pollination; or (ii) state-insensitive feedback. In case (i), snapshots may remain misclassified until the next round of inference. If the new tests are sufficient to show a distinction between the snapshots, then the misclassification will be rectified. In case (ii), the fuzzer does not have sufficient information to make a distinction, since the different behaviors do not yield different observable effects. The choice of a state-sensitive feedback metric is thus important. Through sufficient testing and state-sensitive feedback, state inference always yields accurate groupings.

Comparision to LibAFL: Similar to Tango, LibAFL offers building blocks for custom fuzzers. However, LibAFL is designed for coverage-guided greybox fuzzing but not for stateful fuzzing. Tango is developed independently of and in parallel with LibAFL and was built from the ground up with state as an anchor. Statefulness could be refitted on LibAFL but with heavyweight modification to support for target state as the context of its operations.

8 Related Work

State-aware fuzzing: AFLNet [24] was among the first to tackle the problem of fuzzing network targets while allowing state to accumulate. However, its requirement to manually annotate server responses hindered adoption. Alternative techniques presented in SGFuzz [5], NSFuzz [25], and StateAFL [20] addressed this issue through more automated, albeit less precise techniques for state extraction. Nonetheless, those works focused on extracting state, not as a way to generate protocol-compliant inputs, but as a labeling mechanism for discovered inputs. State inference in Tango makes this mechanism more explicit by exploring functional overlaps. LLM-guided fuzzing [19] has also shown its efficacy at targeting network protocols, by providing machine-readable grammars and high-quality seeds for covering states and transitions. Nonetheless, it remains limited to the information available in its training data and struggles to generalize to arbitrary or proprietary protocols.

Seed scheduling: While seed scheduling is a well-researched problem in fuzzing [12, 29, 30], state inference is the first to address it in the context of stateful systems. Existing techniques do not account for persistent effects of executing seeds; they perform their analysis retrospectively. In contrast, state inference runs a prospective analysis, finding overlaps in the traces of inputs executed.

Grammar inference: State inference overlaps with the disciplines of regular language grammars and automata theory. DFA minimization [13] proposes a technique for collapsing a FMS into a minimum number of states distinguishable by their outgoing transitions. The RPNI[22] and L* [2] algorithms present techniques for passive and active learning of deterministic finite automata for regular languages. Recent work by Luo et al. [17] also tackles grammar inference for network protocols, which is limited to protocols that reveal stateful information in their responses to client messages.

Though similar to DFA inference, our approach differs in key assumptions, filling gaps in existing techniques. Namely, we do not assume an FSM model for the target, nor that the complete set of inputs and responses is known or enumerable. It is not the goal of state inference to extract the state model of the system, since that would require knowledge of which computational model is implemented, e.g., FSM or PDA. State inference instead aims to discover hidden capabilities of each snapshot through cross-pollination then finds groups of snapshots that share the same capabilities through subsumption and colorization. Compared to MACE [8] which first introduced blackbox state inference into dynamic symbolic execution, Tango observes more fine-grained feedback via instrumentation and has addressed unique challenges when combining state inference with seed scheduling.

9 Conclusion

Research on stateful fuzzing continued where its stateless counterpart left off. While much of the progress on the latter was of great benefit to this field, it still managed to imprint methods and assumptions that are otherwise not suited for stateful fuzzing. In this paper, we re-assess the definition of states and how they fit into the fuzzing stack. We present a method to identify semantic behavior through the use of portable metrics, in a technique we dub "State Inference". In the process, we design and implement Tango, a state-aware fuzzing framework for bootstrapping research in this domain. Through evaluation, we identify a key observation: fuzzers could potentially spend upwards of 86% of their time being tunnelvisioned or duplicating their efforts. By applying our technique, fuzzers can leverage state awareness for more optimal scheduling, at a diminishing amortized cost. State inference is also applicable in other stages of the fuzzing cycle, from seed minimization and distillation, through unsupervised state extraction and determined reproduction, to better-grounded performance evaluation.

Acknowledgments

We thank the anonymous reviewers for their feedback on the paper. This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2_186974, and a gift from Huawei.

References

[1] 2022. libfaketime modifies the system time for a single application. https://github.com/wolfcw/libfaketime Accessed: 2024-03-15.

[2] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Information and Computation* 75, 2 (1987).

[3] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. IJON: Exploring Deep State Spaces via Fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*.

[4] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. 2002. The Nonstochastic Multiarmed Bandit Problem. *SIAM J. Comput.* 32, 1 (2002).

[5] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful Greybox Fuzzing. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security '22)*. USENIX Association.

[6] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the Exponential Cost of Vulnerability Discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery.

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery.

[8] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. 2011. {MACE}:{Model-inference-Assisted} concolic exploration for protocol and vulnerability discovery. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security '11)*. USENIX Association.

[9] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security '21)*. USENIX Association.

[10] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT '20)*. USENIX Association.

[11] Andrea Fioraldi, Alessandro Mantovani, Dominik Maier, and Davide Balzarotti. 2023. Dissecting American Fuzzy Lop: A FuzzBench Evaluation. *ACM Transactions on Software Engineering and Methodology* (2023).

[12] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2021*. ACM.

[13] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Chapter Section 4.4.3, Minimization of DFA's, 159–164.

[14] S. Kullback and R. A. Leibler. 1951. On Information and Sufficiency. *The Annals of Mathematical Statistics* 22 (1951).

[15] Junqiang Li, Senyi Li, Gang Sun, Ting Chen, and Hongfang Yu. 2022. SNPSFuzzer: A Fast Greybox Fuzzer for Stateful Network Protocols Using Snapshots. *IEEE Transactions on Information Forensics and Security* (2022).

[16] Chung-Yi Lin, Chun-Ying Huang, and Chia-Wei Tien. 2019. Boosting Fuzzing Performance with Differential Seed Scheduling. In *2019 14th Asia Joint Conference on Information Security (AsiaJCIS)*.

[17] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. 2023. BLEEM: Packet Sequence Oriented Fuzzing for Protocol Implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*.

[18] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998).

[19] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large Language Model-guided Protocol Fuzzing. In *Network and Distributed System Security Symposium (NDSS 2024)*. The Internet Society.

[20] Roberto Natella. 2022. StateAFL: Greybox Fuzzing for Stateful Network Servers. *Empirical Software Engineering* 27, 7 (2022).

[21] Roberto Natella and Van-Thuan Pham. 2021. ProFuzzBench: A Benchmark for Stateful Protocol Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[22] J. Oncina and P. García. 1992. *Pattern Recognition and Image Analysis.* Series in Machine Perception and Artificial Intelligence, Vol. 1. World Scientific, Chapter Inferring Regular Languages in Polynomial Updated Time, 49–61.

[23] Sriram V. Pemmaraju and Steven S. Skiena. 2003. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica.* Cambridge University Press, Cambridge, England. 231–234 pages. Contracting Vertices, Section 6.1.1.

[24] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: A Greybox Fuzzer for Network Protocols. In *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*.

[25] Shisong Qin, Fan Hu, Bodong Zhao, Tingting Yin, and Chao Zhang. 2022. NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing. *International Fuzzing Workshop (FUZZING) 2022* (2022).

[26] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *30th USENIX Security Symposium (USENIX Security 21)*.

[27] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2022. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*.

[28] C. E. Shannon. 1948. A Mathematical Theory of Communication. *Bell System Technical Journal* 27 (1948), 379–423, 623–656.

[29] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*.

[30] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. In *28th Annual Network and Distributed System Security Symposium (NDSS 2021)*. The Internet Society.

A Subsumption Operators

To formalize the approach, we propose the definitions:

$N^{\square}(u)$
: The $\square$-*neighborhood* of u, where $\square \in \{+, -\}$ represents outward and inward directions, respectively.

$u \preceq^{\square} v \iff N^{\square}(u) \subseteq N^{\square}(v)$
: u is $\square$-*subsumed* by v.

$u \prec^{\square} v \iff (u \preceq^{\square} v) \wedge (v \not\preceq^{\square} u)$
: u is *strictly* $\square$-*subsumed* by v.

$u \preceq v \iff (u \preceq^{+} v) \wedge (u \preceq^{-} v)$
: u is *subsumed* by v.

$u \prec v \iff (u \preceq^{+} v) \wedge (u \prec^{\square} v)$
: u is *strictly subsumed* by v, for any $\square \in \{+, -\}$.

$u \sim^{\square} v \iff (u \preceq^{\square} v) \wedge (v \preceq^{\square} u)$
: u and v are $\square$-*equivalent.*

$u \sim v \iff (u \sim^{+} v) \wedge (u \sim^{-} v)$
: u and v are *equivalent.*

For each proposed operator op, we can construct a set of vertices $u \in U$ which satisfy $u\ op\ v$ as: $op_v = \{u \in U | u\ op\ v\}$. Of particular interest are the equivalence sets $\sim_v$ and strict subsumption sets $\prec_v$.

To extract the sets of equivalence states from a capability matrix, it suffices to construct the set of out-equivalence sets:

$$\tilde{S} = \{\sim_v^{+} | v \in L\}$$

Each element in $\tilde{S}$ represents an equivalence state, a set of snapshots that overlap in behavior. To further reduce the number of states, the fuzzer calculates the set of non-empty strict subsumption sets:

$$\check{S} = \{\prec_v \neq \phi | v \in L\}$$

Following that, any snapshot belonging to any set in $\check{S}$ can be safely eliminated from all sets in $\tilde{S}$, without loss of capabilities. Alternatively, to avoid inadvertently losing quality inputs, we choose to include strictly subsumed snapshots in the equivalence states of the subsuming nodes. This ensures that the fuzzer experiences the same reduction in state counts while maintaining access to all interesting snapshots.

B Time Spent on Cross-Pollination

During cross-pollination, for every batch of m new snapshots, given n existing batches, the fuzzer must perform additional executions on the order of $O(m \times n)$.

In particular, we define the following quantities:

Time-step t_k: the point in time at which the kth application round of state inference is performed.

Snapshots $n_k = n_{k-1} + m = mk$

States $s_k = s_{k-1} + \alpha\, m = \alpha mk$

average reduction ratio ↑ additional executions per round ↓

Cross-tests $c_k = c_{k-1} + m(2n_{k-1} + m - 1) = n_k^2 - n_k$

Since coverage growth is linear in exponential time [6], during the kth step, at time t_k, the total number of cross-tests performed is $O(\log^2(t_k))$. Meanwhile, the fuzzer generates and tests new inputs in linear time, diminishing the ratio of time spent on state inference as:

$$\lim_{t\to\infty} \frac{\log^2(t)}{t} = 0$$

C Cost Reduced by State Broadcast

Since discovered capabilities are then broadcast to all snapshots within a state, the number of cross-tests becomes:

$$\begin{aligned}\tilde{c}_k &= \tilde{c}_{k-1} + m(n_{k-1} + s_{k-1} + m - 1)\\ &= mk\left[\frac{\alpha+1}{2}m(k+1) - \alpha m - 1\right]\end{aligned}$$

The approximate reduction in total cross-tests is then:

$$(1-\alpha)\frac{k-1}{2k}; \text{when } \frac{1}{m} \ll 1$$

D Built-in Extensions

Components in Tango are left open for customization, to accommodate for arbitrary stateful systems beyond network services. The framework also encourages re-usability by providing components with fixed interfaces, as well as a dynamic component discovery subsystem to ensure that specialized co-dependent modules are instantiated together. Through its supplementary profiling module, Tango also enables the instrumentation of the fuzzer's own functions and variables, to improve the debuggability of the fuzzer's operations and to better attribute feature changes to improvements in performance.

D.1 AsyncIO

Async I/O is a form of cooperative scheduling, where the application specifies when control is returned to the scheduler. We built Tango as an asynchronous application to enable graceful suspension of the fuzzer and extend its compatibility to event-driven systems, such as DOOM.

D.2 Hotpluggable Inference

We implemented state inference both as a strategy for use in Tango and as a plug-in to third-party fuzzers such as AFL++ and Nyx-Net. We slightly augment the scheduling routines of those fuzzers to incorporate the inference results generated by Tango during a fuzzing campaign and provide them with state-specific feedback.

D.3 Built-in Extensions

Tango ships with a set of complementary modules that enable it to fuzz x86_64 processes on Linux-based systems, reload state through replayed inputs, measure and classify SanitizerCoverage feedback, communicate over standard file descriptors and network sockets, train state-specific mutators and perform state inference.

D.4 `ptrace`-d processes

To ensure synchronicity between the fuzzer and the running process, we use `ptrace` with `seccomp` filters to place catchpoints over the relevant I/O syscalls where necessary, e.g., `read`, `dup`, `close`, and `poll`, among others. Synchronization allows the target to return control to the fuzzer as soon as it becomes ready, instead of busy-waiting and degrading throughput. Moreover, it guarantees reproducibility of results: by encoding the relevant sequences of syscalls into its saved inputs, the fuzzer can reliably reproduce states and coverage measurements, leaving little for the OS to influence when data is delivered to the target.

In addition, to increase fuzzer throughput and exploit the redundancy of resets, Tango leverages `ptrace` to dynamically inject a forkserver at runtime, just after setting up the communication channel in the target. This relieves the loader of the heavy initialization phase of many network services.

D.5 Container isolation

Fuzzing is an embarrassingly parallel process, and it is commonly employed by launching multiple concurrent campaigns on capable machines. When fuzzing network services, this can introduce the problem of overlapping socket bind addresses, since a server's configuration parameters are often identical across campaigns. We therefore isolate each fuzzer instance in a Linux network namespace, allowing them to communicate with their target without aliasing other instances.

We also leverage mount namespaces to achieve filesystem isolation. Some targets may store persistent state through local files, influencing other instances of the target across resets. To avoid that, we mount an `overlay` filesystem on top of a `tmpfs` mount point for storing instance-local data. Then, upon reset, we clear the `upper` filesystem of the `overlay`, effectively destroying any persistent state left by the target.

D.6 Record-and-replay

Tango ships with a default loader which implements a record-and-replay mechanism for loading snapshots. Under the reasonable assumption that the target is deterministic, such a loader can reliably reproduce paths by relaunching or forking the target and re-applying a saved input. More sophisticated snapshot-ing methods exist [27] which can be ported for use under Tango; however, this remains out-of-scope of the current extensions.

D.7 SanitizerCoverage

In our study on state inference, we primarily model features as code coverage profiles, classified into AFL-style bins. To achieve that, Tango implements a CoverageTracker which sets up a shared memory region for communicating coverage updates and extracting feature sets. The tracker is equipped with C-based bindings for performing the binning and hashing with minimal impact on the fuzzer's critical path.

D.8 `socket+stdio`

Tango includes a demonstrative set of channels, to communicate with the target over network sockets, such as TCP and UDP, as well as standard input. These channels extend the `ptrace` functionality

to synchronize the state of the socket or file in the target with its counterpart in the fuzzer. By capturing syscalls such as `bind` and `accept`, we inject a forkserver at the latest stage in initializing the target. This achieves around a 50x increase in fuzzing throughput over the non-optimized implementation since socket setup is often expensive, and otherwise, the fuzzer may only successfully connect to the target by reattempting the operation until it no longer fails.

D.9 Adaptive mutators

We implement an adaptive model for applying havoc mutators that balances exploration and exploitation using the Exp3 [4] algorithm for distributing rewards and assigning probabilities. For each snapshot, we maintain a set of weights describing the probability that a mutator is chosen in that state. Provided a comprehensive set of mutators, this approach accommodates an evolving target by adjusting and applying the probabilities in selecting the next mutator, based on how well each one performs in the state context.

E Case Study: DOOM

Tango is a framework for building state-aware fuzzers, and the main witness of its merit would be using it to develop a state-aware fuzzer for a complex stateful system. In the spirit of hacker culture, we opted to answer the question "Can it run DOOM?" with a resounding "Yes!", using Tango.

E.1 Setup

To support DOOM, we extended Tango in the following aspect:

Driver: We implemented an `X11Channel` component which sends keystroke events to a process's window, through a public Python library, `python3-xlib`.

Generator: We added `Activate`, `Kill`, `Move`, `Reach`, `Rotate`, and `Shoot` instructions, extending the input base class. These govern how the player's character interacts with its environment, and are later used by the input generator and the exploration strategy to maneuver around the map and overcome obstacles.

We limited the functionality of the input generator to selecting a possible outgoing or incoming transition of the current state and passing it on to the mutator, which mainly mutates the direction and duration of movement, to explore the level. We also provided it with two helper functions that can yield the correct sequence of commands to follow a path or to aim and shoot at a moving target by incorporating live feedback.

Tracker: We implemented state feedback as a shared-memory struct populated by DOOM and accessed by Tango. The struct contains all basic user properties such as location, weapons, ammo, pickups, enemies in sight, and doors or switches within reach. Two states are considered equivalent if they have the same player position. We attached extended state variables to each state that describe the pickups collected along the current path, and state attributes describing the current location (e.g. if it is a slime pit or a secret level). To avoid having a unique state for every single position on the map, the level is divided into a grid of cells, representing the granularity of feedback, as shown in Fig. 14a.

Loader: We extended the loader's two main functions: restarting the target and loading a state. Restarts are simple, as they're only a matter of terminating and relaunching the process. Loading a state is slightly more complex: without a means of snapshotting, actions must be replayed to reach a certain known location, given that the state graph contains at least one path to it. However, a downside of this approach is that nearby states (locations) may be reachable through a bee-line movement, whereas the input generated and discovered by the fuzzer to transition between these two states may involve redundancies that impact fuzzer throughput. Another downside is that if the current location and the target location are close to each other, yet are far enough from the spawn location, restarting the level from that location would be inefficient. The player may simply need to move a few steps in the direction of the target to reach it. Moreover, continuously restarting the target breaks the immersion of the fuzzer "playing" the game, and it would instead spend much of its time replaying actions from the start. To avoid that, we implemented path-finding algorithms, based on the state graph explored by the fuzzer, to move between two locations using a sequence of `Reach` instructions. In essence, to load a state, a path to it from the current state is calculated, and `Reach` instructions are performed piece-wise along every transition in the path.

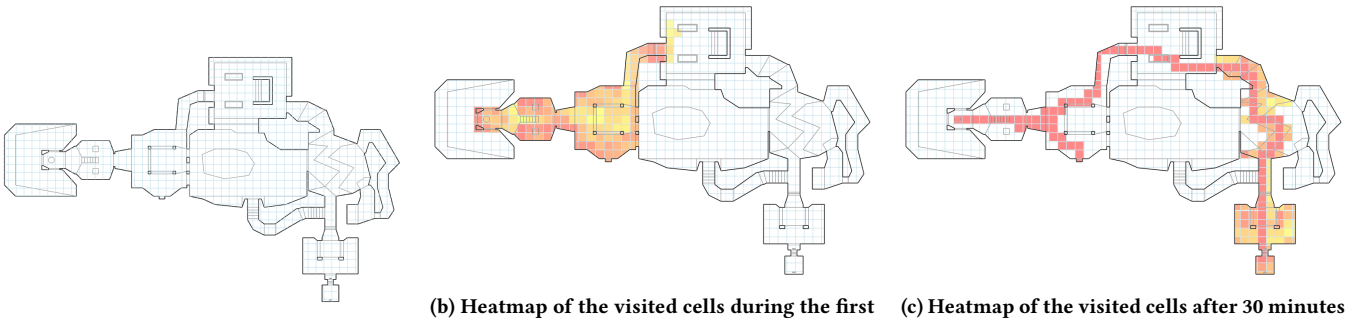
Strategy: Finally, we implemented a `ZoomStrategy` component, to tie it all together, that schedules states based on a convex hull of the explored locations, and prioritizes locations on the perimeter, that are furthest from the start. In addition, the strategy implements an event observer task that is responsible for reacting to urgent events such as seeing an enemy or stepping in slime pits. By preempting the fuzzer's main loop, the strategy minimizes the reaction time to increase the survivability of the player.

E.2 Results

With these extensions, Tango consistently manages to finish the E1M1 level of DOOM, on difficulty 3, in 10 to 40 wall-clock minutes. The main factor contributing to this variability is perimeter exploration. As can be seen in Fig. 14b, in one recorded fuzzing session, the fuzzer encountered a big undiscovered area on the left side of the map, and dedicated a significant amount of time to exploring it. It also managed to discover the path up the stairs to the higher platform, where it found a level 1 armor pickup. In other runs, due to the stochastic nature of the fuzzing process, it may miss that area completely and continue exploring in the immediate direction of the exit door, achieving a lower overall finish time in the process.

Having found the armor pickup, Tango maintains a record of it in its later exploration stages. As can be seen in Fig. 14c, its path to the finish line includes going up the stairs, picking up the armor boost, and returning back on another path to the exit room.

Regardless of the overall finish time, once Tango finds the path to the exit, it consistently manages to follow it in 3 to 4 in-game minutes. While far from typical speed-runs for this level (which are as low as 9 seconds), it remains a formidable achievement to be able to explore the state space of a DOOM level and manage to finish it in a sensible amount of time.

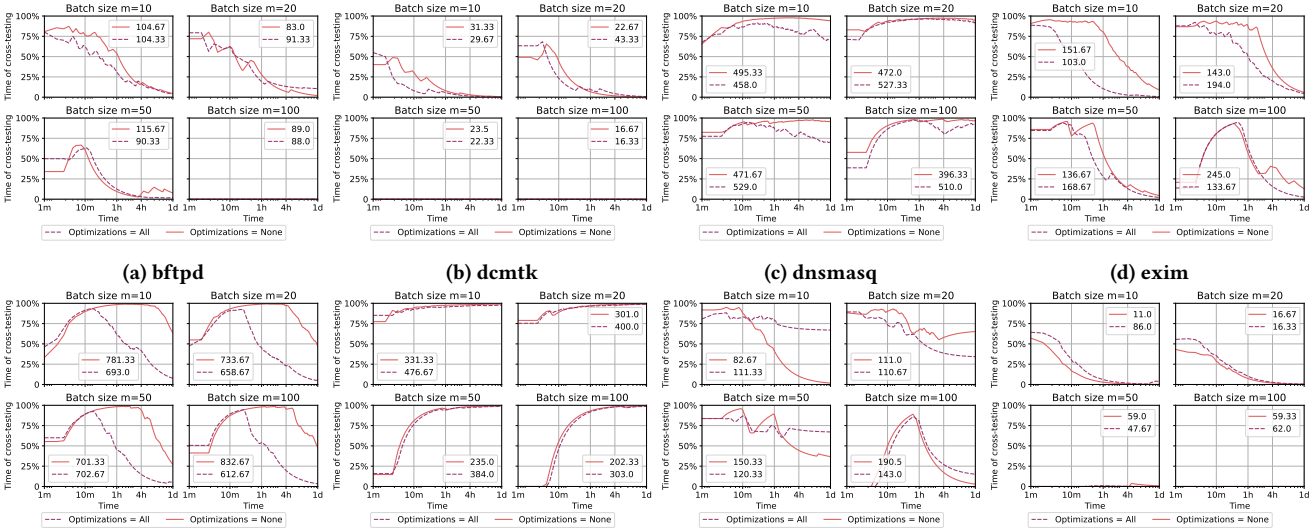


(a) Grid view of the DOOM map.

(b) Heatmap of the visited cells during the first 10 minutes of fuzzing DOOM's the E1M1 level.

(c) Heatmap of the visited cells after 30 minutes of fuzzing.

Figure 14: The progress of TANGO in playing DOOM. Red shading implies higher hit counts. Figure 14c shows that the fuzzer had figured out a path to the finish and continues to repeat it to achieve the lowest completion time.



(a) bftpd (b) dcmtk (c) dnsmasq (d) exim

(e) expat (f) kamailio (g) lightftp (h) live555

(i) llhttp (j) openssh (k) openssl (l) proftpd

(m) pureftpd (n) tinydtls (o) yajl

Figure 15: The time of cross-testing per target.