# IJON: Exploring Deep State Spaces via Fuzzing

Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz
Ruhr University Bochum

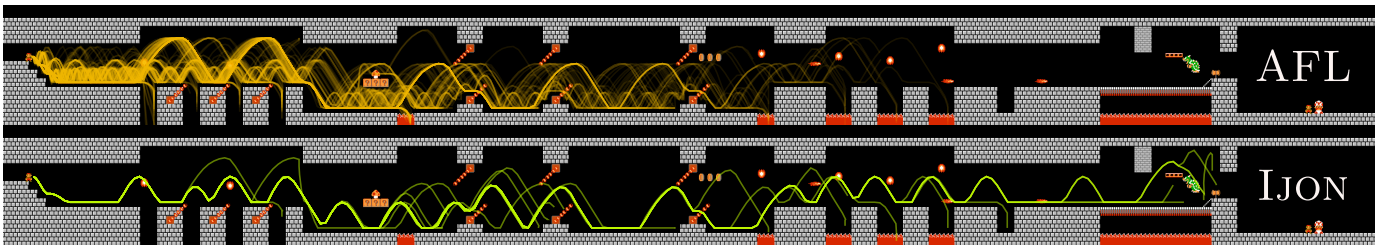



Fig. 1: AFL and AFL + IJON trying to defeat Bowser in Super Mario Bros. (Level 3-4). The lines are the traces of all runs found by the fuzzer.

***Abstract*—Although current fuzz testing (*fuzzing*) methods are highly effective, there are still many situations such as complex state machines where fully automated approaches fail. State-of-the-art fuzzing methods offer very limited ability for a human to interact and aid the fuzzer in such cases. More specifically, most current approaches are limited to adding a dictionary or new seed inputs to guide the fuzzer. When dealing with complex programs, these mechanisms are unable to uncover new parts of the code base.**

**In this paper, we propose IJON, an annotation mechanism that a human analyst can use to guide the fuzzer. In contrast to the two aforementioned techniques, this approach allows a more systematic exploration of the program's behavior based on the data representing the internal state of the program. As a consequence, using only a small (usually one line) annotation, a user can help the fuzzer to solve previously unsolvable challenges. We extended various AFL-based fuzzers with the ability to annotate the source code of the target application with guidance hints. Our evaluation demonstrates that such simple annotations are able to solve problems that—to the best of our knowledge—no other current fuzzer or symbolic execution based tool can overcome. For example, with our extension, a fuzzer is able to play and solve games such as *Super Mario Bros.* or resolve more complex patterns such as hash map lookups. To further demonstrate the capabilities of our annotations, we use AFL combined with IJON to uncover both novel security issues and issues that previously required a custom and comprehensive grammar to be uncovered. Lastly, we show that using IJON and AFL, one can solve many challenges from the CGC data set that resisted all fully automated and human guided attempts so far.**

## I. INTRODUCTION

In recent years, a large number of software bugs were uncovered by fuzz testing (short: *fuzzing*) and this research area has received significant attention in both the academic community [7], [14], [43], [45], [53], [60] and practice [1], [33], [61]. As a result, much attention was placed on further improving fuzzing methods, often to achieve greater code coverage and reach deeper into a given software application. Yet, a significant number of open challenges remain: Even with clever program analysis techniques such as symbolic or concolic execution, some constraints cannot be overcome easily. Furthermore, in some cases, state explosion proves too much of a hindrance to current techniques—whether they are fuzzing or symbolic execution based approaches. This is due to the fact that the underlying problem (finding bugs) is undecidable in the general case. As a result, we cannot expect that any single algorithm will perform very well across all target applications that are tested.

Due to this insight, even though significant progress has been made in recent works on improving fully autonomous fuzzers, some constraints will remain unsolvable no matter which algorithm is used (e.g., if cryptography is used). In practice, current approaches struggle to explore complex state machines, where most progress can only be observed in changes to the program's state data. Since each update to the state data is triggered by certain code, a coverage-based fuzzer is able to explore each individual update in isolation. However, there is no feedback that rewards exploring *combinations* of different updates leading to new states, if all individual updates have been observed previously. In cases where a specific sequence of updates is needed to uncover a bug, this prevents the fuzzer from making progress. Similarly, concolic execution based approaches fail, since the exact sequence of updates (and consequently the precise code path chosen) is critical to uncover the bug. Since concolic execution fixes the path to the observed execution path, it is impossible for the solver to obtain an input that triggers the target condition. Lastly, even fully symbolic execution, which is free to explore different paths, fails if the state space grows too large.

We note that there is a trend to use more complex solutions, which only support minimal environments/instruction sets, based on symbolic execution to overcome harder challenges in fuzzing [43], [45], [53], [60]. On the downside, as observed by various sources [10], [60], [62], such methods sometimes scale poorly to complex applications. As a result, they find little use in industry, compared to fuzzers such as LIBFUZZER and AFL. Google's OSS fuzz project alone was able to uncover over 27.000 bugs [23] in targets as complex as the Chrome browser using tools such as LIBFUZZER. Often, it seems, the additional



DOI 10.1109/SP40000.2020.00117

effort to set up and deal with a symbolic environment is not worth the effort [62].

In this paper, we explore how a human can steer the fuzzer to overcome current challenges in fuzzing. To paraphrase a well-known, but hard to attribute quote, "Computers are incredibly fast, accurate and stupid; humans are incredibly slow, inaccurate and brilliant; together they are powerful beyond imagination". Humans are often better at forming high-level, strategic plans, while a computer ensures that the tactics are working out, and the human does not overlook any important aspect. This approach is typically referred to as *human-in-the-loop*, and is a commonly used concept in software verification [8], [15], [34], [39], [41], [52] and throughout various other fields such as machine learning [17], [59], controlling cyber-physical systems [47], [49], and optimization [24], [51].

Our approach is also motivated by the observation that many fuzzing practitioners in the industry already use a closed feedback loop in their fuzzing process [35]: First, they run the fuzzer for some time and then analyze the resulting code coverage. After this manual analysis, they tweak and adapt the fuzzing process to increase coverage. Common strategies for improving the fuzzing performance include removing challenging aspects from the target application (e.g., checksums), changing the mutation strategies, or explicitly adding input samples that solve certain constraints that the fuzzer did not generate in an automated way. This approach has two main reasons: on the one hand, all the "easy" bugs (i.e., the ones which can be found fully automatically) are found very quickly during a fuzzing campaign. On the other hand, the more interesting bugs are—by definition—the ones that cannot be found using current tools in off-the-shelf configurations and hence, some manual tuning is required. We believe that by assisting and steering the fuzzing process, humans interacting with fuzzers allow for a vastly increased ability to analyze applications and overcome many of the current obstacles related to fuzzing of complex applications.

Specifically, we focus on a particular class of challenges: we observe that current fuzzers are not able to properly explore the state space of a program beyond code coverage. For example, program executions that result in the same code coverage, but different values in the state, cannot be explored appropriately by current fuzzers. In general, the problem of exploring state is challenging, as it is difficult to automatically infer which values are interesting and which are not. However, a human with a high-level understanding of the program's goals, often knows which values are relevant and which are not. For example, a human might know that exploring different player positions is relevant to solve a game, while the positions of all enemies in the game world are not.

We show that a human analyst can annotate parts of the state space that should be explored more thoroughly, hence modifying the feedback function the fuzzer can use. The required annotations are typically small, often only one or two lines of additional code are needed. To demonstrate the practical feasibility of the proposed approach, we extended various AFL-based fuzzers with the ability to annotate the source code of the target application with hints to guide the fuzzer. Our extension is called IJON, named after Ijon Tichy, the famous space explorer from Stanislaw Lem's books [31]. In four case studies, we show that the annotations can help to overcome significant roadblocks and to explore more interesting behaviors. For example, using simple annotations, we are able to play *Super Mario Bros.* (as illustrated in Figure 1) and to solve hard instances from the CGC challenge data set.

In summary, we make the following contributions in this paper:

- We systematically analyze feedback methods implemented in current fuzzers, study how they represent state space, and investigate in which cases they fail in practice.
- We design a set of extensions for current feedback fuzzers that allow a human analyst to guide the fuzzer through the state space of the application and to solve hard constraints where current approaches fail.
- We demonstrate in several case studies how these annotations can be used to explore deeper behaviors of the target application. More specifically, we show how the state space of a software emulator for a Trusted Platform Module (TPM), complex format parsers, the game *Super Mario Bros.*, a maze, and a hash map implementation can be efficiently explored by a fuzzer. Additionally, we demonstrate, that our approach enables us to solve some of the most difficult challenges in the CGC data set, and find new vulnerabilities in real-world software.

The implementation of IJON and a complete data set that illustrates our evaluation results, including playthrough videos for the case studies we present in this paper, is available at https://github.com/RUB-SysSec/ijon.

## II. TECHNICAL BACKGROUND

Our work builds upon fuzzers from the AFL [61] family such as ANGORA [14], AFLFAST [11], QSYM [60], or LAF-INTEL [1]. To explain the technical details of our approach, we hence need to introduce some aspects of the inner working of AFL itself. Fuzzers from the AFL family generally try to find a corpus that triggers a large variety of different states from the state space of the program. Here, a *state* denotes one configuration of memory and registers, as well as the state provided by the OS (e.g., file descriptors and similar primitives). The *state space* is the set of all possible states a program can be in. Since even for trivially small programs, the state space is larger than the number of atoms in the universe, the fuzzer has to optimize the diversity of states reached by the test cases. This class of fuzzers typically uses code coverage to decide if an input reaches sufficiently different state than the ones existing in the corpus. We make use of the *bitmap* that AFL uses to track this coverage. Hence, we start by explaining the design of the coverage feedback used by AFL. Afterward, we discuss some of the consequences of this design when using a fuzzer to optimize the state coverage.

### A. AFL *Coverage Feedback*

Various forks of AFL use instrumentation to obtain test coverage information. Typically, AFL-style fuzzers track how

often individual edges in the control flow graph (CFG) are executed during each test input. There are two classes of feedback mechanisms commonly used: source code based forks of AFL typically use a custom compiler pass that annotates all edges with a custom piece of code. Binary-only versions of AFL use different mechanisms such as Dynamic Binary Instrumentation (DBI) or hardware accelerated tracing (typically Intel LBR or Intel PT) to obtain coverage information. Either way, the probes inserted into the target application then count the occurrences of each edge in the CFG and store them in a densely encoded representation.

The resulting information is stored in a *shared map* that accumulates all edge counts during each test run. The fuzzer additionally maintains a *global bitmap* that contains all edge coverage encountered during the whole fuzzing campaign. The global bitmap is used to quickly check if a test input has triggered new coverage. AFL considers a test input interesting and stores it if it contains a previously unseen number of iterations on any edge. Since edges always connect two basic blocks, edges are encoded as a tuple consisting of two identifiers, one for the source basic block $id_s$ and one for a target block $id_t$. In the source code based versions, a static random value is assigned at compile-time to each basic block, which is used as $id_s$ or $id_t$. For binary-only implementations, it is common to use a cheap hash function applied to the address of the jump instruction/target instruction to derive the $id_s$ and $id_t$ values, respectively. This tuple $(id_s, id_t)$ is then used to index a byte in the shared map. Typically, the index is calculated as $(id_s * 2) \oplus id_t$. The multiplication is used to efficiently distinguish self-loops.

Before each new test run, the shared map is cleared. During the test run, each time an edge is encountered, the corresponding byte in the shared map is incremented. This implies that edge counts greater than 255 will overflow and might register as any number between 0 and 255. After the execution finished, the edge counts are bucketed such that each byte in the shared map with a non-zero edge count contains a power of 2. To this end, edge counts are discretized into the following ranges $1, 2, 3, 4 \ldots 7, 8 \ldots 15, 16 \ldots 31, 32 \ldots 127, 128 \ldots 255$. Each range of edge counts is assigned to one specific power of 2. To increase the precision on uncommon edges, 3 also maps to a unique power of 2, while the range 32 to 64 is omitted. Then we can compare the shared map against a global bitmap, which contains all bits that were previously observed in prior runs. If any new bit is set, the test input is stored because it has led to increased coverage, and the global bitmap is updated to contain the new coverage.

### B. Extending Feedback Beyond Coverage

Fuzzers sometimes get stuck in a part of the search space, where no reasonable, probable mutation provides any new feedback. In this paper, we develop novel ways to provide a smoother feedback landscape. Consequently, we now review various methods that were proposed to extend feedback mechanism beyond code coverage to avoid getting stuck on a plateau. Notably, LAF-INTEL [1] was an early approach to solve magic byte type constraints (e.g., `if (input == 0xdeadbeef)`) by splitting large compare instructions into multiple smaller ones. The same idea was later implemented by using dynamic binary instrumentation in a tool called STEELIX [32]. Splitting multi-byte compare instructions into multiple single byte instructions allows the fuzzer to find new coverage every time a single byte of the operands is matching. ANGORA [14] assigns a random identifier to each function. It uses these identifiers to extend the coverage tuple by a third field that contains a hash of the current execution context. To compute this hash, it combines all identifiers of functions that have been called but have not yet returned (i.e., active functions) using an XOR operation. This allows finding the "same" coverage if it was used in a different calling context. For example, this method is helpful to solve multiple calls to the `strcmp` function. However, the downside of this approach (i.e., considering all calling contexts as unique) is that in certain situations, this creates a large number of inputs that are actually not interesting. Therefore, ANGORA requires a larger bitmap than AFL.

## III. DESIGN

Generally speaking, a fuzzer tries to sample from "interesting" regions of the state space as efficiently as possible. However, it is hard to find an accurate and objective metric for how "interesting" the state space of any given input is. The overall success of AFL and its derivative demonstrates that following the edge coverage is an effective metric to identify new interesting regions. Edge coverage is probably the feature with the best signal to noise ratio in practice—after all, in most (i.e., not obfuscated [25], [30]) programs, each new edge indicates a special case. However, there are code constructs where this approach is unlikely to reach new coverage *without* exploring intermediate points in the state space. In the following, we analyze code constructs in which a user could provide additional feedback that would help the fuzzer by conceptually describing the intermediate steps which provide additional feedback. Finally, we introduce a novel set of primitives that actually allow an analyst to add custom annotations which provide exactly the feedback needed to overcome these difficulties.

### A. State Exploration

To identify problematic code constructs that are hard to fuzz with current techniques, we performed several offline experiments using state-of-the-art fuzzers and manually inspected the coverage obtained. In some cases, we used seed files to find code that can be covered using good seeds, but not without, indicating hard constructs. In the following, we summarize the most important problems we encountered in these experiments:

- *Known Relevant State Values*: Sometimes, code coverage adds no feedback to help the fuzzer to advance. If only a small subset of the states is interesting and a human analyst is able to identify these values, we can directly use them to guide the fuzzer.

- *Known State Changes*: Sometimes, the program is too complex, or it is not obvious which variables contain interesting state and which ones do not. In such situations, since no sufficiently small set of relevant state values are known to us, we cannot directly use them to guide the fuzzer. Instead, a human analyst might be able to identify positions in the code that are suspected to mutate the state. An analyst can use the history of such state changes as an abstraction for the more complex state and guide the fuzzer. For example, many programs process messages or chunks of inputs individually. Processing different types of input chunks most likely mutates the state in different ways.
- *Missing Intermediate State*: Unlike the previous two cases, there might be neither variables that contain the state, nor code that mutates the state that we care about. In such situations, an analyst can create artificial intermediate states to guide the fuzzer.

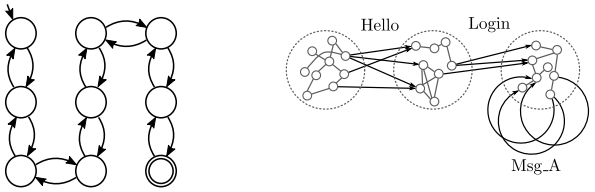
Based on this systematization of important problems in fuzzing, we provide examples and describe each of the suggested approaches in more detail.

*1) Known Relevant State Values:* As described before, sometimes the code coverage yields nearly no information about the state of the program, because all the interesting state is stored in data. For example, the coverage tells very little about the behavior of a branch-free AES implementation. If the analyst has an understanding of the variables that store the interesting state, he can directly expose the state to the fuzzer and the fuzzer is then able to explore inputs that cause different internal states.

```
while(true) {
    ox=x; oy=y;

    switch (input[i]) {
        case 'w': y--; break;
        case 's': y++; break;
        case 'a': x--; break;
        case 'd': x++; break;
    }

    if (maze[y][x] == '#'){ Bug(); }

    //If target is blocked, do not advance
    if (maze[y][x] != ' ') { x = ox; y = oy; }
}
```

Listing 1: A harder version of the maze game.

Consider the code in Listing 1. It implements a small game, in which the player has to navigate a labyrinth by moving in one of four possible directions. It is based on the famous labyrinth demo that is commonly used by the symbolic execution community to demonstrate how a symbolic executor can explore the state space of a maze. In this modified version, it is possible to walk backward and to stay in the same place. This creates a vast amount of different paths through the program. At the same time, there are effectively only four branches that can be covered, thus the coverage alone is not a good indicator of interesting behavior. In this harder version, even KLEE [12] fails to solve the labyrinth. Here, it is essential to understand that the $x$ and $y$ coordinates are relevant states that need to be explored. In mazes with dead ends, it is even impossible to find a solution by trying to increase $x$ or $y$ individually. The combination of both $x$ and $y$ has to be considered to uncover a solution. Since the maze is rather small (at most only a few hundred different $x, y$ pairs are reachable), the analyst can instruct the fuzzer to consider any new pair as new coverage (Figure 2.a).




(a) The abstract state machine of the maze game reduced to $x, y$ pairs. Since the mazes have only a few states, the fuzzer can enumerate all.

(b) The analyst can abstract the large set of states into smaller set of states that are supposedly separated by certain operations. The fuzzer can then explore edge tuples on this state machine.

Fig. 2: An analyst view on two fuzzing problems.

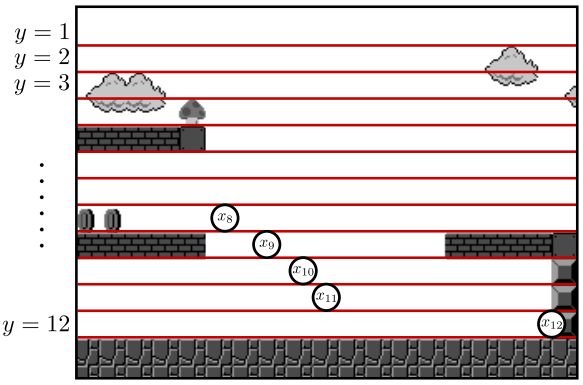



Fig. 3: The circles mark the best observed $x$ values so far. Optimizing $x$ for each altitude independently helps overcoming dead ends. In this case, the input maximizing the value observed at $y = 12$ ($x_{12}$) is very unlikely to be helpful (Mario will hit the wall or the ceiling). However, the inputs with the best observed values for other altitudes (such as $x_{10}$) are not caught in a dead end.

Similar scenarios, where the user is aware of the precise aspect of the state that is interesting to explore also occur with larger state spaces. One example that demonstrates a similar scenario is the game Super Mario Bros. (Figure 3). Again, we mostly care about the player coordinates. However, the space of player coordinates is significantly larger (in the order of the $10^6$) than in the maze game. As a consequence, the analyst needs to be able to provide a goal that the fuzzer can work towards (i.e., increase the $x$ coordinate), instead of simply exploring all different states. This way, the fuzzer can discard inferior intermediate results.

*2) Known State Changes:* In some scenarios, the user might not be aware of which parts of the relevant state are interesting to explore or the state might be spread out across

the application and hard to identify. Alternatively, the state might be known, but no sufficiently small subset can be identified to be used to guide the fuzzer directly. However, it might be possible to identify parts of the code that are expected to change the state. This situation commonly occurs in applications that consume highly structured data such as sequences of messages or lists of chunks in file formats. In such situations, instead of directly exposing the state itself, the user can create a variable that contains a log of messages or chunk types. This variable can act as a proxy for the actual state changes and can be exposed to the feedback function (Figure 2.b). As a consequence, the fuzzer can now try to explore different combinations of those state changes. In such scenarios, the state change log serves as an abstraction layer to the real state that cannot easily be exposed to the fuzzer. We further elaborate on this in Example 4 of Section III-D.

```
msg = parse_msg();
switch(msg.type) {
    case Hello: eval_hello(msg); break;
    case Login: eval_login(msg); break;
    case Msg_A: eval_msg_a(msg); break;
}
```

Listing 2: A common problem in protocol fuzzing.

Consider the dispatcher code shown in Listing 2, which is based on many common protocol implementations. The fuzzer will successfully uncover the different messages. However, AFL has difficulties to generate interesting sequences of messages, as no novel coverage is produced for chaining messages. The fundamental problem here is that the fuzzer is not able to distinguish between different states in the program state machine. By using a log of the types of messages that were successfully processed, the fuzzer is able to explore different states, resulting from combinations of messages, much more effectively.

*3) Missing Intermediate State:* A simple example for issues where neither coverage nor values in the program provide relevant feedback are magic byte checks. Out of the box, AFL-style fuzzers will not be able to solve them. Note that various approaches try to solve this case using additional methods [1], [7], [14], [53], [60]. However, the same problem persists in more complex cases: if the relationship between the input and the final comparison gets more complex, even techniques like concolic execution will fail. A human analyst on the other hand, can usually reason about how the program behaves and can often provide an indicator of progress. By encoding this indicator as additional *artificial intermediate states*, the analyst is able to guide the fuzzer. Note that for simple magic bytes like situations, this is exactly what LAF-INTEL does.

Consider the code in Listing 3, which is based on a hard case found in the well-known `objdump` binary. The program contains a function that performs a hash table lookup and one `if` condition where the lookup result is used to search for a given string. More specifically, the key is first hashed, and the corresponding bucket is checked. If the bucket is empty, no further processing takes place. If the bucket contains some values, they are compared individually. This poses a significant challenge both to a concolic execution based tools as well as fuzzers: solving this constraint requires to find a very specific combination of both a path (i.e., number of loop iterations necessary to calculate the hash) and the hash value that all share the same coverage. Even if the exact path is found, we now still depend on both the hash and the actual string matching. The fuzzer has to solve the comparison while maintaining that the hash of the input is always equal to the hash of the target string. A concolic executor would have great trouble finding the exact path to solve this constraint.

```
//shortened version of a hashmap lookup from binutils
entry* bfd_get_section_by_name(table *tbl, char *str) {
  entry *lp;
  uint32_t hash = bfd_hash_hash(str);
  uint32_t i = hash % tbl->size;

  //Every hash bucket contains a linked list of strings
  for (lp = tbl->table[i]; lp != NULL; lp = lp->next) {
    if (lp->hash == hash && strcmp( lp->string, str) == 0)
          return lp;
  }
  return NULL;
}

// used somewhere else
section = bfd_get_section_by_name (abfd, ".bootloader");
if (section != NULL){ ... }
```

Listing 3: A hash map lookup that is hard to solve (from binutils libbfd, available at bfd/section.c).

We found that similar conditions occur more than 500 times with various strings throughout the `binutils` code base. In most cases, the hash table is filled with values from the input, and a fixed string is looked up. A human can recognize that this effectively implements a one-to-many string comparison. Using this insight, she can guide the fuzzer to find a solution, by turning this complex constraint into a series of simple string comparisons that can be solved with LAF-INTEL-like feedback.

### B. Feedback Mechanisms

As no current fuzzer allows a human analyst to directly provide feedback to the fuzzer, we design a set of annotations that allow the analyst to influence the fuzzer's feedback function. Our goal is that an analyst can use these annotations to provide high-level steering for the fuzzing process. In an interactive fuzzing session, the analyst inspects the code coverage from time to time to identify branches that seem hard to cover for the fuzzer. Then the analyst can identify the reason why the fuzzer is unable to make progress. Typically, this is an easy task for a human. When the road block is found, the analyst can start a second fuzzing session that focuses on solving this road block using a custom annotation. The annotation itself is a small patch to the target application, typically consisting of one, sometimes two lines of code that provide additional feedback information. When the fuzzer solves the road block, the long-running fuzzing session picks the inputs

that produce new coverage from the temporary session and continues fuzzing, hence overcoming the hard case.

To facilitate a workflow like this, we designed four general primitives that can be used to annotate source code:

1) We allow the analyst to select which code regions are relevant to solve the problem at hand.
2) We allow direct access to the AFL bitmap to store additional values. Bitmap entries can either be directly set or incremented, hence enabling to expose state values to the feedback function.
3) We enable the analyst to influence the coverage calculation. This allows the same edge coverage to result in *different* bitmap coverage. This allows to create much more fine-grained feedback in different states.
4) We introduce a primitive that allows the user to add hill climbing optimization [48]. This way, the user can provide a goal to work towards if the space of possible states is too large to explore exhaustively.

In the following, we explain these annotations in more detail and illustrate how they work and can be used to implement the additional feedback mechanisms described in Section III-A.

### C. IJON-*Enable*

The IJON-ENABLE (and IJON-DISABLE) annotation can be used to enable and disable coverage feedback. This way, we can effectively exclude certain parts of the code base or guide the fuzzer to only explore code if certain conditions are met.

```
IJON_DISABLE();
//...
if(x<0)
    IJON_ENABLE();
```

Listing 4: Using the IJON-ENABLE annotation. The green highlight indicates an added annotation.

**Example 1.** *Consider the annotation (green highlight) in Listing 4. In this example,* IJON-ENABLE *restricts the temporary fuzzing sessions to inputs that reach the annotated line and have a negative value for* $x$*. This annotation allows the fuzzer to focus on the hard problem without wasting time exploring the many other paths in the input queue.*

### D. IJON-INC *and* IJON-SET

The IJON-INC and IJON-SET annotations can be used to increment or set a specific entry in the bitmap. This effectively allows new values in the state to be considered as equal to new code coverage. The analyst can use this annotation to expose aspects of the state to the fuzzer selectively. As a result, the fuzzer can then explore many different values of this variable. Effectively, this annotation adds a feedback mechanism beyond code coverage. The fuzzer is now also rewarded for *new data coverage* obtained via its test cases. This annotation can be used to provide feedback in all three scenarios that we described earlier.

```
IJON_INC(x);
```

Listing 5: Using the IJON-INC annotation to expose the value $x$ to the feedback function.

**Example 2.** *Consider the annotation shown in Listing 5. Every time* $x$ *changes, a new entry in the bitmap is incremented. For example, if* $x$ *is equal to 5, we calculate an index in the bitmap based on a hash of the current file name and the current line number and the value 5. The corresponding entry in the bitmap is then incremented. This allows the fuzzer to learn a variety of inputs that display a large range of behaviors, bit by bit.*

Similarly, to the IJON-INC annotation, we also provide the IJON-SET annotation. Instead of incrementing the entry, this annotation sets the least significant bit of the bitmap value directly. This enables to control specific entries in the bitmap to guide the fuzzer. This primitive is used in all three annotation approaches introduced earlier.

```
while(true) {
    ox=x; oy=y;

    IJON_SET(hash_int(x,y));
    switch (input[i]) {
        case 'w': y--; break;
//....
```

Listing 6: Annotated version of the maze.

**Example 3.** *We added a one-line annotation (see Listing 6) to the maze game introduced in Section III-A1. It uses the combination of both* $x$ *and* $y$ *coordinates as feedback. As a result,* any *newly visited position in the game is considered as new coverage. We used* IJON-SET *instead of* IJON-INC*, since we do not care how often the given position was visited. Instead, we are only interested in the fact that a new position was visited.*

If the state cannot easily be observed, we can use state change logging (as described earlier), in which we annotated operations which are known to affect the state that we care about and use the log of state changes as index for the feedback.

**Example 4.** *As another example, consider Listing 7. After each successfully parsed and handled message, we append the command index, which represents the type of the message, to the state change log. Then we set a single bit, addressed by the hash of the state change log. As a consequence, whenever we see a new combination of up to four successfully handled messages, the fuzzer considers the input as interesting, providing a much better coverage in the state space of the application.*

### E. IJON-STATE

If the messages cannot easily be concatenated (e.g., because there is a message counter), the state change log might be

```
//abbreviated libtpms parsing code in ExecCommand.c
msg = parse(msg);
err = handle(msg);
if(err != 0){goto Cleanup;}

state_log=(state_log<<8)+command.index;
IJON_SET(state_log);
```

Listing 7: Annotated version of libtpms.

insufficient to explore different states. To produce a more fine-grained feedback, we can explore the Cartesian product of the state and the code coverage. To enable this, we provide a third primitive that is able to change the computation of the edge coverage itself. Similar to ANGORA, we extended the edge tuple with a third component named the "virtual state". This virtual state component is also considered when calculating the bitmap index of any edge. This annotation is called IJON-STATE. Whenever the virtual state changes, any edge triggers new coverage. This primitive has to be used carefully: if the number of virtual states grows too large, the fuzzer is overwhelmed with a large number of inputs which effectively slows down the fuzzing progress.

```
IJON_STATE(has_hello + has_login);
msg = parse_msg();
//...
```

Listing 8: Annotated version of the protocol fuzzing example (using IJON-STATE).

**Example 5.** *Consider the example provided in Listing 8. As discussed previously, without annotations, the fuzzers may have difficulties exploring the combination of various messages. By explicitly adding the protocol state to the fuzzer's virtual state, we create multiple virtual copies of the code, depending on the protocol state.*

*Therefore, the fuzzer is able to fully explore all possible messages in various states of the protocol state machine. Effectively, the same edge coverage can now result in different bitmap coverage, and hence the fuzzer can efficiently explore the state space of the program under test. Note that, to prevent state explosion, there are only three possible values for the state. As a result, the fuzzer can fully re-explore the whole code base, once successfully authenticated.*

### F. IJON-MAX

So far, we mostly dealt with providing feedback that can be used to increase the diversity of the inputs stored. In some cases, however, we want to optimize towards a specific goal or the state space is simply too large to cover completely. In such cases, we might not care about a diverse set of values or want to discard all intermediate values. To allow effective fuzzing in such cases, we provide a maximization primitive called IJON-MAX. It effectively turns the fuzzer into a generic hill climbing-based black box optimizer. To enable maximizing more than one value, multiple (by default 512) slots are provided to store those values. Like the coverage bitmap, each value is maximized independently. Using this primitive, it is also possible to easily build a minimization primitive for $x$, by maximizing $-x$.

```
//inside main loop, after calculating positions
IJON_MAX(player_y, player_x);
```

Listing 9: Annotated version of the game Super Mario Bros.

**Example 6.** *Consider the video game* Super Mario Bros.*, in which a player controls a character in a side-scrolling game. In each level, the objective is to reach the end of the level, while avoiding hazards such as enemies, traps, and pits. In case the character is touched by an enemy or falls into a pit, the game ends. To properly explore the state space of the game, it is important to reach the end of each level. As illustrated in Listing 9, we can finish the level by asking the fuzzer to try to maximize the player's $x$ coordinate. Given that it is a side-scrolling game, this effectively guides the fuzzer to find a way through the level to successfully finish it.*

*The* IJON-MAX *(slot,x) annotation then tells the fuzzer to maximize the $x$ coordinate of the character. Note that we use the player's $y$ coordinates (height) to select the slot. This allows us to maximize the progress at different altitudes within the level independently. By increasing the diversity in the set of inputs, we reduce the chance of getting stuck in a dead end as shown in Figure 3. Using this technique, we can quickly find solutions for 29 out of 32 levels of the game. More details are available in Section V-C.*

## IV. IMPLEMENTATION

We implemented IJON as an extension for multiple fuzzers whose implementation is based on AFL: AFLFAST, LAF-INTEL, QSYM, and ANGORA. All of these fuzzers share the same underlying code base, and thus, the required changes to implement our method were similar for all fuzzers. Overall, we performed two different kinds of changes. On the one hand, we implemented a way to apply annotations to the target application. On the other hand, we extended the communication channel between IJON and the target.

### A. Adding Annotations

To enable coverage feedback, AFL comes with a special compiler pass for `clang` that instruments every branch instruction. Additionally, AFL provides a wrapper that can be used instead of `clang` to compile the target. This wrapper automatically injects the custom compiler pass. We extended both the wrapper and the compiler pass. To support our changes, we introduced an additional runtime library that the compiler links statically. The runtime implements various helper functions and macros that can be used to annotate the target application. In particular, we added support for fast hash functions that can be used to generate better-distributed values or compress strings to an integer. In summary, we used the primitives from Section III and added some more high-level helper functions.

*1)* IJON-ENABLE*:* To implement IJON-ENABLE, we introduce a mask that is applied to all bitmap index calculations. If the mask is set to zero, only the first bitmap entry can be addressed and updated. If it is set to `0xffff`, the original behavior is used. This way, we can effectively disable and enable coverage tracing at will.

*2)* IJON-INC *and* IJON-SET*:* Both annotations enable direct interaction with the bitmap, hence the implementation is straightforward. Upon a call to `ijon_inc(n)`, the $n^{\text{th}}$ entry in the bitmap is incremented. Similarly, calling `ijon_set(n)` sets the least significant bit of the $n^{\text{th}}$ entry to 1.

If these functions are used in multiple locations within the program code, one has to be very careful not to reuse the same bitmap indices. To help avoid such cases, we introduce helper macros `IJON_INC(m)` and `IJON_SET(m)`. Both macros call the corresponding function but calculate the bitmap index `n` based on a hash of `m` as well as the filename and line number of the macro invocation. Thus, avoiding trivial collisions on commonly used arguments such as 0 or 1.

*3)* IJON-STATE*:* When there is a call to `IJON_STATE(n)`, we change the way basic block edges are mapped to bit map entries. To this end, we change the definition of the edge tuple to include the state in addition to the source ID and target ID: $(\text{state}, id_s, id_t)$. Here, "state" is a thread local variable that stores information related to the current state. Calling `IJON_STATE(n)` updates $\text{state} := \text{state} \oplus n$. That way, two successive calls cancel each other out.

We also modified the compiler pass such that the bitmap index is calculated as follows: $\text{state} \oplus (id_s * 2) \oplus id_t$. This way, every time the state variable changes, every edge gets a new index in the bitmap. These techniques allow the same code to produce different coverage if it is executed in a different context.

*4)* IJON-MAX*:* We extended the fuzzer to maintain an additional, second queue of inputs for maximization purpose. We support to maximize up to 512 different variables. Each of these variables is called a *slot*. The fuzzer only ever stores the input that produces the best value for each slot and discards old inputs that resulted in smaller values. To store the largest observed value, we introduce an additional shared memory *max-map* consisting of 64-bit unsigned integers. Calling the maximization primitive `IJON_MAX(slot,val)` updates `maxmap[slot]=max(maxmap[slot], val)`.

After executing a test input, the fuzzer checks both the shared bitmap and the max-map for new coverage. Similar to the design of a shared coverage bitmap and the global bitmap (as explained in Section II-A), we also implemented a global max-map that persists during the whole fuzzing campaign and complements the shared max-map. In contrast to the bitmap, no bucketing is applied to the shared max-map. An entry in the shared max-map is considered novel if it is larger than the corresponding entry in the global max-map.

Since we now have two queues of inputs, we must also update our scheduling strategy. IJON asks the user to provide a probability for using the original queue (generated from code coverage) or the maximization queue (generated from maximizing slots). This functionality lets the user decide which queue has more weight on picking inputs to fuzz. The user can supply the environment variable `IJON_SCHEDULE_MAXMAP`, with a value from zero to 100. Each time a new input is scheduled, the fuzzer draws a random number between one and hundred. If the random number is smaller than the value of `IJON_SCHEDULE_MAXMAP`, the usual AFL based scheduling takes place. Otherwise, we pick a random non-zero slot in the max-map and fuzz the input corresponding to that slot. If the same slot is updated while fuzzing its input, the old input is immediately discarded, and the fuzzing stage is continued based on the newly updated input.

*5) Helper Functions:* The runtime library of IJON contains a set of helper functions which can simplify common annotations. For example, the runtime library contains helper functions to hash different kinds of data such as strings, memory buffers, the stack of active functions, or the current line number and file name. Additionally, there are helper functions which can be used to calculate the difference between two values. We implemented different helper functions to simplify the annotation process, as described below:

- **`IJON_CMP(x,y)`**: computes the number of bits that differ between $x$ and $y$. This helper function directly uses it to touch a single byte in the bitmap. It is worth mentioning that for practical purposes, it is not directly using the number of different bits as an index to the bitmap. Consider Listing 5, if the same annotation was reused in multiple locations, the indices would collide (both share the range 0 to 64). Instead, `IJON_CMP` combines the argument with a hash of the current file name and line. Thus, we drastically reduce the chance of a collision.
- **`IJON_HASH_INT(u32 old, u32 val)`**: returns a hash of both **`old`** and **`val`**. As described in Section III-D, we use hash functions to create more diverse set of indices and to reduce the probability of a collision.
- **`IJON_HASH_STR(u32 old, char* str)`**: returns a hash of both arguments. For example, we use this helper function to create a hash of the file name.
- **`IJON_HASH_MEM(u32 old, u8* mem, size_t len)`**: returns a hash of **`old`** and the first **`len`** bytes of **`mem`**
- **`IJON_HASH_STACK (u32 old)`**: returns a hash of the return addresses of active functions (which we call *execution context*). All addresses are hashed individually. The resulting values are XORed together to produce the final result. That way, recursive calls are not creating too many different values. This helper function can be used to create virtual copies of the feedback based on the hashed execution context.
- **`IJON_STRDIST(char* a, char* b)`**: evaluates and returns the length of the common prefix of `a` and `b`.

### B. Communication Channel

To communicate the intermediate coverage results, AFL-based fuzzers use a shared memory region containing the shared bitmap. We extend this region by replacing it with a shared struct. This struct has two different fields. The first one is the original AFL shared bitmap. The second one is the shared max-map used for our maximization primitive.

## V. Evaluation

As noted earlier, we implemented IJON on top of five state-of-the-art fuzzing tools. This way, we show how the same annotation helps to outperform all of the baseline fuzzers according to various metrics. We picked multiple different targets to show how we can overcome a variety of hard problems using only a small number of annotations.

### A. Setup

For each experiment, we compare the unmodified fuzzer against the fuzzer with annotations. All experiments are performed on a Debian 4.9.65-3 machine with an Intel Xeon E5-2667 processor with 12 cores and the clock speed of 2.9 GHz, plus 94 GB of RAM. Unless noted otherwise, a single uninformative seed containing only the character "a" was chosen. The use-case for IJON is to have whatever fuzzer available to run until (mostly) saturated, inspect the coverage, and improve it using manual annotations. As a consequence, in some experiments, we pick small examples and assume that we only care about solving this individual hard aspect. This simulates that we use `IJON_ENABLE` to limit fuzzing to the interesting area. The baseline fuzzers would typically be unable to focus on a single area and perform far worse at that specific task, since they also explore other parts of the target application. Since we compare against the baseline fuzzers on the isolated example, we conservatively strengthen the position of the unaided fuzzers.

### B. The Maze – Small, Known State Values

The maze is a famous example that is often used to demonstrate the power of symbolic execution based test case generation. It consists of a simple game similar where a player has to walk through an ASCII art maze. The publicly available version is straightforward due to two factors: The maze contains no dead ends, and every move but the correct one immediately terminates the game. Out-of-the-box AFL and similar fuzzers are not able to solve this maze in a reasonable amount of time, while KLEE can find a solution in a few seconds. The effective state space of this game is linear in the number of steps taken, and no state explosion takes place. To make this problem more interesting, we created a harder version of the game. In the hard version, the player can walk backward and stay at the same place (by walking into a wall) without dying. As a result, the state space grows exponentially in the number of steps taken. It turns into a complete tree of size $4^n$. Now KLEE is unable to solve the maze as well. Additionally, we created a larger labyrinth including dead ends. Note that while the ideas implemented in IJON are also applicable to the search heuristics used by KLEE, we did not implement IJON on top of KLEE. We still included an unmodified version of it in our evaluation. The results of our experiments are shown in Table I and Table II.

*a) Initial Run:* We performed experiments on both the small and the large maze. Each experiment was executed three times using two configurations. In the first configuration, we use the original, easy rules, where any wrong move immediately terminates the game. In the second configuration, a harder rule set is used. As a consequence of the harder rules, the game has an exponentially larger state space. Each fuzzer was tested both in the unmodified version and in combination with IJON. Each experiment was conducted for one hour. Since different tools spawn sub-processes and threads, all tools were locked to one core; it should be noted that AFL can occasionally guess the solution for the smallest maze within one hour. However, no coordinated progress is made after the first few minutes, and therefore, we reduced the experiment duration to one hour.

*b) IJON:* We added a simple one-line annotation (Listing 10) that uses the combination of both $x$ and $y$ coordinates as feedback. Any newly visited position in the game is treated like new coverage.

```
while(true) {
    ox=x; oy=y;

    IJON_SET(hash_int(x,y));
    switch (input[i]) {
        case 'w': y--; break;
//....
```

Listing 10: Annotated version of the maze game.

As Table I shows, none of the tools is able to solve the more complex variants of the maze. Yet, with a single, one-line annotation and the support of IJON, all fuzzers can solve all maze variants. Table II shows the time it took for different fuzzers to solve the maze variant. The fuzzers which could not solve the maze are excluded from the table.

Using IJON annotations, AFL performs quite comparable to KLEE on the easy maze. On the hard version, AFL with IJON is even faster than KLEE on the easy version. Note that KLEE is unable to solve the hard maze. Additionally, while AFL without IJON is sometimes able to solve the small maze in the easy mode, AFL in combination with IJON is more than 20 times faster than AFL without IJON. Lastly, most extensions of AFL such as LAF-INTEL, QSYM and ANGORA are actually decreasing the performance compared to baseline AFL. This is due to the additional mechanisms provided by these fuzzers, which incur an extra cost while not providing any value in this scenario.

### C. Super Mario Bros. – Large Known State Values

Another instance of a similar problem is the game *Super Mario Bros.* We modified the game in such a way that all keyboard commands are read from `stdin`. Additionally, we modified the game such that the game character Mario always

TABLE I: Different approaches are solving the small / large maze. Three runs were performed each. We show the number of solves for the small / large mazes. An ✗ denotes no solution was found in any runs, a ✓ indicates that all runs solved the maze. Lastly, we could not extend KLEE and hence there are no results for KLEE with IJON.

| Tool | Plain | | IJON | |
|---|---|---|---|---|
| | Easy Small / Large | Hard Small / Large | Easy Small / Large | Hard Small / Large |
| AFL | $\frac{2}{3}$ / ✗ | ✗ / ✗ | ✓ / ✓ | ✓ / ✓ |
| AFLFAST | ✗ / ✗ | ✗ / ✗ | ✓ / ✓ | ✓ / ✓ |
| LAF-INTEL | ✗ / ✗ | ✗ / ✗ | ✓ / ✓ | ✓ / ✓ |
| QSYM | $\frac{1}{3}$ / ✗ | ✗ / ✗ | ✓ / ✓ | ✓ / ✓ |
| ANGORA | ✗ / ✗ | ✗ / ✗ | $\frac{1}{3}$ / $\frac{2}{3}$ | $\frac{2}{3}$ / ✓ |
| KLEE | ✓ / ✓ | ✗ / ✗ | | |

TABLE II: Different approaches are solving the small/large maze. The table shows the average time-to-solve in minutes ± the standard deviation.

| Tool | Easy | Hard |
|---|---|---|
| AFL-plain | $42.7 \pm 11.9$ / - | - / - |
| QSYM-plain | $50.7 \pm 0.0$ / - | - / - |
| KLEE-plain | $0.7 \pm 0.5$ / $2.0 \pm 0.0$ | - / - |
| AFL-ijon | $1.8 \pm 1.0$ / $7.6 \pm 3.3$ | $0.5 \pm 0.2$ / $2.3 \pm 1.2$ |
| AFLFAST-ijon | $1.6 \pm 0.5$ / $8.4 \pm 1.5$ | $0.5 \pm 0.1$ / $5.7 \pm 2.4$ |
| LAF-INTEL-ijon | $2.3 \pm 0.9$ / $7.6 \pm 1.5$ | $0.7 \pm 0.3$ / $3.4 \pm 2.3$ |
| QSYM-ijon | $5.4 \pm 1.6$ / $11.4 \pm 1.4$ | $5.3 \pm 0.1$ / $11.3 \pm 0.6$ |
| ANGORA-ijon | $42.4 \pm 0.0$ / $36.5 \pm 0.3$ | $7.5 \pm 0.2$ / $15.9 \pm 3.8$ |

runs to the right, and Mario is killed when he stops moving. This decision was made to produce speed run like results, with short execution times. We use both AFL and AFL in combination with IJON annotations to play the game and observe the progress made in the game.

We ran AFL without annotations for 12 hours and observed how far into the different levels Mario was able to reach. We added a simple annotation that uses the `ijon_max` primitive. We create one slot for each possible height (measured in tiles). In each slot, we maximize the $x$ coordinate of the player. The changed code was shown earlier in Listing 9. The results can be seen in Table III, an example was shown earlier in Figure 1. With this simple one line annotation, the fuzzer solved nearly all levels in a matter of minutes. In fact, using IJON, AFL is able to solve all but 3 levels. It should be noted that only one of these levels (level 6-2) can be solved. The fact that it remained unsolved is due to the inherent randomness in fuzzing. In another offline experiment, IJON was sometimes able to solve the level in less than 6 hours. Level 4-4 is unsolvable due to a bug in the emulation. The last level seems impossible/extremely hard to solve due to the modifications we made in the game.

Plain AFL, on the other hand, struggles: It takes significantly longer to solve far fewer levels and makes less progress in the levels it did not solve. Note that AFL is in fact surprisingly good. On the first look, it might seem that AFL should have nearly no feedback to uncover as it advances throughout the levels. Yet, it appears that the edge counts for events such as "spawn a pipe, cloud, enemy, or bush" is already good enough to solve about one-third of the levels.

*a) Glitches:* Super Mario Bros. contains various well-known glitches and secret passages, commonly exploited by speed-runners. Our fuzzer was able to find and exploit at least two of those. First, we were able to exit the top of the level in stage 4-2, leading to the famous warp zone. Second, at various times Mario used a "wall-jump" and escape certain deaths. To perform a wall-jump, Mario has to jump into a wall, landing exactly at the corner of a tile. For one frame, Mario's foot enters the wall, before the physics engine pushes Mario out of the wall again. In this frame, Mario is able to jump again. This trick requires both perfect timing, as well as very precisely hitting the corner of a wall tile (Mario's position is tracked with sub-pixel accuracy).

TABLE III: AFL and IJON playing Super Mario Bros. (3 experiments each). We show how often the level was successfully solved, the median time until the best input was found (hh:mm) and the median percentage of distance traveled. Videos from the best runs are available at https://github.com/RUB-SysSec/ijon. Note that in level 4-2, IJON is able to uncover the secret room.

| | AFL | | | IJON | | |
|---|---|---|---|---|---|---|
| Level | Solved | Time | % Distance | Solved | Time | % Distance |
| 1-1 | $\frac{1}{3}$ | 07:11 | 91% | ✓ | 00:25 | 100% |
| 1-2 | ✗ | 07:12 | 61% | ✓ | 00:59 | 100% |
| 1-3 | $\frac{1}{3}$ | 08:39 | 71% | ✓ | 00:20 | 100% |
| 1-4 | $\frac{2}{3}$ | 08:37 | 100% | ✓ | 00:23 | 100% |
| 2-1 | ✗ | 09:55 | 42% | ✗ | 06:44 | 94% |
| 2-2 | ✗ | 07:31 | 57% | ✓ | 00:16 | 100% |
| 2-3 | ✗ | 09:25 | 88% | ✓ | 00:18 | 100% |
| 2-4 | ✗ | 00:31 | 65% | ✓ | 00:06 | 100% |
| 3-1 | ✗ | 08:16 | 33% | ✓ | 02:42 | 100% |
| 3-2 | ✗ | 07:15 | 80% | ✓ | 00:28 | 100% |
| 3-3 | $\frac{2}{3}$ | 08:36 | 100% | ✓ | 00:03 | 100% |
| 3-4 | $\frac{1}{3}$ | 09:35 | 77% | ✓ | 00:40 | 100% |
| 4-1 | ✗ | 05:41 | 83% | ✓ | 00:11 | 100% |
| 4-2 | ✗ | 02:07 | 95% | ✓ | 02:06 | 118% |
| 4-3 | ✓ | 11:24 | 100% | ✓ | 00:16 | 100% |
| 4-4 | ✗ | 02:22 | 82% | ✗ | 00:06 | 82% |
| 5-1 | ✗ | 11:11 | 60% | ✓ | 00:45 | 100% |
| 5-2 | ✗ | 11:27 | 73% | ✓ | 00:23 | 100% |
| 5-3 | $\frac{2}{3}$ | 09:19 | 100% | ✓ | 00:26 | 100% |
| 5-4 | ✓ | 04:32 | 100% | ✓ | 00:12 | 100% |
| 6-1 | $\frac{1}{3}$ | 08:56 | 93% | ✓ | 00:36 | 100% |
| 6-2 | ✗ | 09:57 | 36% | ✗ | 06:32 | 82% |
| 6-3 | $\frac{2}{3}$ | 06:01 | 100% | ✓ | 00:08 | 100% |
| 6-4 | $\frac{2}{3}$ | 06:01 | 100% | ✓ | 00:07 | 100% |
| 7-1 | ✗ | 04:44 | 63% | ✓ | 00:25 | 100% |
| 7-2 | ✗ | 06:37 | 58% | ✓ | 00:13 | 100% |
| 7-3 | ✗ | 10:08 | 92% | ✓ | 00:24 | 100% |
| 7-4 | ✓ | 08:07 | 100% | ✓ | 00:06 | 100% |
| 8-1 | ✗ | 01:36 | 21% | $\frac{2}{3}$ | 06:02 | 100% |
| 8-2 | ✗ | 07:17 | 60% | $\frac{2}{3}$ | 03:12 | 100% |
| 8-3 | $\frac{2}{3}$ | 07:18 | 100% | ✓ | 00:18 | 100% |

### D. Structured Input Formats – State Change Logging

To demonstrate the ability to explore structured input formats using state change logging, we evaluate AFL with IJON on three examples. The first two are taken from evaluation of AFLSMART [44]. In their paper, Pham et al. present multiple case studies; in particular, the paper discusses two examples (`libpng` and `WavPack`) in more detail. We picked both examples and added a simple `state_change_log` annotation as described in Section III-A2. We ran both annotated versions

for 24 hours on an Intel Core i7-6700 CPU clocked at 3.40 GHz with 16 GB of RAM. We also provided the first 1024 bytes of a random `wav`/`png` file as seed and used a dictionary of strings from the source code. With this simple annotation, we were able to uncover bugs in both targets, while AFL did not find any of the bugs. Additionally, we found another out-of-bounds read in `libpng` that is still present (but reported) in the most up-to-date version of `libpng`. This demonstrates that for these structured input formats, a grammar is not actually needed to discover these security issues. Adding a small annotation in the parser code can sometimes provide the exact same benefit at much smaller cost to the analyst.

To further substantiate this observation, we picked another example to demonstrate how AFL with the same `state_change_log` annotation explores a much larger combination of features than plain AFL would. LIBTPMS is a software emulator for Trusted Platform Module (TPM) security co-processors. It is commonly used in virtualization solutions to create virtualized modules. TPM specifies a protocol that can be used to interact with the co-processor. The protocol consists of a wide variety of possible messages, each of which has a specific parser. This situation reassembles Listing 2. We demonstrate that AFL only explores a small subset of all possible message sequences. Then we introduce a single IJON annotation (two lines). In AFL, we now observe a massive increase in the number of combinations of messages explored. Note that we used AFL's persistent mode to improve performance of the fuzzing and obtain more sound results within the allocated times. However, other tools such as ANGORA and QSYM are incompatible with this approach. This is less problematic, since in this experiment we do not care about solving individual constraints. Instead, we care about the number of distinct message sequences explored which neither ANGORA nor QSYM provides a better feedback for this kind of scenario. As a consequence, we chose to only evaluate AFL and AFL with IJON.

*a) Initial Run:* We ran AFL for 24h and plotted the subset of the paths found. The results can be seen in Table IV in the columns labeled "plain". To visualize, we also included the graph of all messages found for one run. Each node is a message that was successfully parsed and handled. If we successfully handled message A and message B after each other, we add an edge between node A and B.

*b) IJON:* We added a change log based annotation that stores the last 4 successfully parsed and handled message IDs. Then we use these IDs to create additional feedback. The annotated code is displayed in Listing 7. Using this annotation, we performed another 24-hour run. The number of paths found can be seen in Table IV in the columns labeled "IJON". We observe, that IJON leads to significantly more complex interactions.

From both the experiments on the AFLSMART examples and on TPM, it can be seen that using state change log annotation drastically increases the ability to explore different chains of messages (and therefore state) that would not be covered using current techniques. In particular, our experiments on `libpng` and `WavPack` demonstrate that this previously unexplored space contains critical bugs that would be much harder to find otherwise.

TABLE IV: The number of distinct message sequences found by AFL and AFL + IJON fuzzing TPM

| Seq. Length | AFL Plain | AFL +IJON | Improvement |
|---|---|---|---|
| 1 | 14 | 14 | 1x |
| 2 | 53 | 156 | 2x |
| 3 | 85 | 1636 | 18x |
| 4 | 79 | 2532 | 32x |

```
// callback used to iterate the hash map
void ijon_hash_feedback(bfd_hash_entry* ent, char* data){
    IJON_SET(IJON_STRDIST(ent->string, data));
}

//shortened version of a hashmap lookup from binutils
entry* bfd_get_section_by_name(table *tbl, char *str) {
//perform a string feedback for each entry in the hashmap.
 bfd_hash_traverse(tab, ijon_hash_feedback, str);
//....  rest of the function as shown earlier.
}
```

Listing 11: Annotated version of the hash map example.

### *E. Binutils Hash map - Missing Intermediate States*

In this final experiment, we extracted the relevant code from the hash map example introduced earlier (Listing 3). To provide an IJON annotation, we use an available iteration function on the hash map. For each entry in the hash map, we perform an IJON string compare. The resulting modification can be seen in Listing 11. In this case, the annotation is somewhat more tricky than the previous annotations. We encode the domain knowledge that the hash map lookup is an efficient version of a one-to-many string comparison. We create an explicit loop of string compare operations in the lookup function. Then we try to maximize the maximum number of matching bytes amongst all compares.

Similar to the maze example, we performed experiments with various fuzzers with and without IJON custom annotations. Since we extracted the hard code from the application, we observed that no new inputs were found after as few as 10 to 20 minutes. As a result, we choose to evaluate each configuration with three experiments of one hour each. The results are displayed in Table V and Table VI. Note that ANGORA is excluded from this experiment as we were unable to compile `libbfd` with ANGORA's taint tracking.

*a) Initial Run:* During the initial runs, none of the fuzzers in the experiment were able to solve the constraints in the unmodified form.

*b) IJON:* We extended the target with a small annotation (three lines, as explained above). After applying this annotation, all fuzzers are able to solve the constraint in a matter of a few minutes, only AFLFAST failed in two out of three runs. However, in its only successful run it managed to solve the constraint in 6 minutes.

TABLE V: Different approaches exploring the hash example. Using IJON, all fuzzers were able to solve the hash at least once. Without IJON, none of the fuzzers could solve this example.

| Tool | Plain | IJON |
|---|---|---|
| AFL | ✗ | ✓ |
| AFLFAST | ✗ | $\frac{1}{3}$ |
| LAF-INTEL | ✗ | ✓ |
| QSYM | ✗ | ✓ |

TABLE VI: Different approaches solving the hash example (average time in minutes ± standard deviation). Note that we only report the numbers for IJON, since the base fuzzers never found a solution.

| Tool | IJON |
|---|---|
| AFL | $8.1 \pm 6.4$ |
| AFLFAST | $6.2 \pm 0.0$ |
| LAF-INTEL | $26.1 \pm 14.8$ |
| QSYM | $15.5 \pm 2.4$ |

### F. Cyber Grand Challenge

To further substantiate that a manual interaction with the fuzzing target is helpful in uncovering security issues, we performed an experiment on the well-known Linux port [2] of the DARPA CGC [3] data set. This also allowed us to differentiate our approach from HACRS [52], another human-in-the-loop approach. We ignored all challenges that could already be solved by AFL, LAF-INTEL [1], REDQUEEN [7], QSYM [60], T-FUZZ [43], or VUZZER [45], as there would be no need for manual analysis when an existing fully automated tool is able to produce a crash. In total, 62 challenges remained undefeated. We picked a random subset of 30 CGC targets. For eight of these targets, even the provided Proof of Vulnerability (PoV) did not cause a crash. This is most likely due to the fact that we used the TRAIL OF BITS x86 Linux port of the challenges [2]. We use this port since our tooling does not support DECREE targets. We inspected each of the 22 remaining challenge manually and performed fuzzing experiments using AFL and IJON. We managed to produce crashes in 10 targets as demonstrated in Table VII. More technical details are available in the appendix.

We were unable to crash the remaining 12 targets due to a variety of reasons: Three targets contained only information leak vulnerabilities that do not cause a crash. Consequently, they are outside of the scope of this work. Two of the targets contained crashes that were too complex to uncover (requiring to trigger a series of different bugs to cause an observable crash). While it would be possible to design IJON annotations that trigger these crashes, we doubt that one could do it without prior knowledge of the bug. Consequently, we count these two targets as failures. Of the remaining 7 targets, two misbehaved in our coverage tracer (crashing on nearly all inputs), three required very large inputs that AFL is not able to generate efficiently, and two were running very slowly or causing timeouts on all but very few inputs. Due to the fact that—unlike HACRS—IJON is aimed at experts, we could only perform this experiment with a single human analyst.

### G. Real-World Software

Beside the memory corruptions we found in the state log experiment in Section V-D and the novel crashes found in the CGC dataset, we also performed experiments on other software to further demonstrate that IJON is useful for finding security bugs. In particular, we picked `dmg2img`, a tool that was very recently fuzzed by the authors of WEIZZ [18]. We applied patches for the vulnerabilities found, and continued fuzzing using IJON. In total, we uncovered three additional memory corruption vulnerabilities in `dmg2img`. Two were variants of the bugs found by WEIZZ, where additional constraints needed to be satisfied to reach other unsafe usages of dangerous string manipulation functions. The third bug is an integer overflow that causes an allocation of size zero. Later on, the byte at offset minus one is set to zero, corrupting malloc metadata as illustrated in Listing 12.

TABLE VII: Comparing IJON with HACRS, another Human-in-the-loop approach, on CGC targets. Note that none of the following CGC targets were solved by state-of-the-art fuzzers. Note that NRFIN_00012 contains a broken random number generation that made the challenge impossible to solve. After fixing the random number generator, we were able to solve the challenge.

| Target | Type | Coverage (#Lines) AFL | Coverage (#Lines) IJON | Crash found HACRS | Crash found IJON |
|---|---|---|---|---|---|
| CROMU_00011 | txt | 70% | 82% | ✗ | ✓ |
| NRFIN_00030 | bin | 84% | 84% | ✗ | ✓ |
| NRFIN_00004 | txt | 21% | 98% | ✗ | ✓ |
| NRFIN_00076 | bin | 24% | 48% | ✗ | ✓ |
| NRFIN_00041 | txt | 21% | 27% | ✗ | ✓ |
| CROMU_00020 | bin | 61% | 75% | ✗ | ✓ |
| NRFIN_00005 | txt | 18% | 73% | ✓ | ✓ |
| NRFIN_00012 | bin | 87% | 73% | ✗ | (✓) |
| NRFIN_00038 | txt | 5% | 81% | ✗ | ✓ |
| NRFIN_00049 | txt | 20% | 61% | ✗ | ✓ |

```
IJON_MAX(kolyblk.XMLLength);
plist = (char *)malloc(kolyblk.XMLLength + 1);
if (!plist)
   mem_overflow();
//....
plist[kolyblk.XMLLength] = '\0';
```

Listing 12: Bug #3 in `dmg2img`

## VI. RELATED WORK

After AFL was published about five years ago, its inner workings were analyzed in detail and multiple improvements were proposed. Different scheduling algorithms were proposed to improve fuzzing in various scenarios [10], [11], [13], [46], [56]. The second component of AFL is the input mutation strategy. Again, various improvements were proposed to increase the probability of generating interesting inputs by means of using more information on the target [6], [7], [9], [27], [40], [44] or taint tracking to limit the amount of input data that needs to be mutated [14], [45]. Lastly, AFL observes the programs behavior for each test case and receives feedback on its behavior. Different fully automated techniques were proposed that help improving the performance of this feedback generation [19], [50] or to extend the feedback [1], [29], [32]. To overcome various roadblocks, such as magic bytes, techniques based on concolic execution [20]–[22], [26], [37], [53], [55], [60], [63] and—less commonly—on symbolic execution [16], [38] were proposed. Since fuzzing is very

performance-critical, various aspects of it were optimized by other projects. For example, Xu et al. improved the performance of spawning subprocesses [58]. Various fuzzers used Intel-PT to increase the performance of coverage tracing on binary targets [5], [50]. Lastly, to make fuzzing more applicable it was adopted to targets ranging from firmware [16], [64] over hypervisors [28] and operating systems [4], [50], [54] to neural networks [42], [57]

### A. Human-in-the-Loop Approaches in Fuzzing

Recently, human-in-the-loop fuzzing gained the attention of the research community. Current approaches commonly provide an interactive environment where the humans interactions are used as seeds by the fuzzer. In some scenarios this is very helpful, while in other common fuzzing scenarios, it is very hard to apply. For example, Shoshitaishvili et al. [52] introduced HACRS, a system that allows humans to interact with the target application via an emulated terminal. To help the human, HACRS analyzes the target and provides a list of strings that might be relevant to the application's behavior. If the target provides a text-based interface, a human is often able to figure out how to interact with the target application. However, HACRS does not work when the target application does not consume data in a format easily understood by humans, e.g., binary formats, or the program output does not contain helpful information on the expected input format. Consequently, the authors excluded any program containing binary characters in the provided example scenarios from their evaluation. Similarly, DYNODROID [36] traces a human's interaction with Android applications. In contrast to HACRS, this includes a more diverse set of input interfaces such as system events and swipe gestures. However, the fundamental principle remains the same. In contrast, our approach is based on annotating the code of the target application and does not require to understand the input format.

Our approach requires little understanding of software security or program analysis. Even complex games can typically be won by untrained users. On the other hand, when fuzzing binary file formats or network protocols, even an expert user will have a hard time to come up with novel inputs based on observing only the program output. IJON allows to guide the fuzzer's intrinsic ability to generate inputs to overcome fuzzing roadblocks and to explore a more diverse part of the state space. This approach requires no knowledge or understanding of the input format at all. In fact, we also typically had very limited understanding of the target application, as our annotations allow to provide guidance without deep understanding of the program context. For example, consider the bug shown in Listing 12. The annotation was added without any context or understanding of how `kolyblk.XMLLength` is calculated from the input. Still, IJON was able to trigger the integer overflow after a few minutes of analysis time.

## VII. Discussion and Future Work

In this paper we describe an interactive approach to fuzzing. While this offers new ways to steer the fuzzing process, it requires manual inspection of the test coverage, acquiring an understanding of why a constraint is hard to solve, and manually creating a well-suited annotation. This manual effort comes with a certain cost to the analyst, but our evaluation demonstrates that this approach can overcome several obstacles for current fuzzers. For example, it is clear that neither a grammar nor a dictionary will help a fuzzer to solve Super Mario levels. Additionally, it is not straightforward to find good seed inputs. While in such a case, a record and replay mechanism such as a version of HACRS or DYNODROID applicable to this target might be helpful, in many other cases such as binary formats, it would likely not.

The annotations used in IJON are currently added manually. It would be interesting to design methods that automatically infer annotations only for difficult individual constraints. Some existing fuzzing approaches use a subset of the annotations described by us. For example, LAF-INTEL can be represented by annotating the number of equal bytes in a comparison. Similarly, ANGORA already implements call stack based virtual state for all coverage. However, all those tools use additional feedback indiscriminately. Therefore, they are limited to low information gain feedback mechanisms. Using more precise feedback in a similar manner would result in a much larger queue and decreased performance. Automated techniques to identify IJON annotations could make use of much more powerful annotations than LAF-INTEL and ANGORA, as they are applied sparsely. Finally, right now, the annotations require source access. In principle, there is no reason why similar annotations cannot be implemented for binary-only targets. Integrating IJON with a binary-only fuzzer would be interesting.

## VIII. Conclusion

In this paper, we have shown that a large number of hard problems for fuzzers can be solved by manually inspecting the code coverage during fuzzing and using only a few lines of annotations to guide the fuzzing process. Previously, practitioners in the industry often used to manually pick seed files that solve the coverage or to design custom mutators that solve constraints (for example, by providing dictionaries or grammars). Similarly, it is well documented that practitioners try to remove hard constraints such as checksum manually. We extended this toolkit with another manual but very flexible method: annotations that an analyst can use to guide the fuzzer. By using less than four lines of code, we demonstrated how a large set of problems could be solved that no other automated fuzzing approach is currently able to handle.

## Acknowledgments

This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 39078197. In addition, this work was supported by the European Union's Horizon 2020 Research and Innovation Programme (ERC Starting Grant No. 640110 (BASTION) and 786669 (REACT)) and the German Federal Ministry of Education and Research (BMBF, project HWSec – 16KIS0592K). The content of

this document reflects the views only of their authors. The European Commission/Research Executive Agency are not responsible for any use that may be made of the information it contains.

## APPENDIX

To elaborate on the results from our experiments on the CGC dataset presented in Section V-F, we now provide some explanation of the annotations we used to solve the 10 targets (see Table VIII for details). Unfortunately, the exact amount of time spent on solving individual targets is somewhat hard to measure, as small changes are often followed by minutes to hours of fuzzing with no human interaction needed. Consequently, we can only report very rough estimates for the time spent on the implementation. In particular, for the shorter examples ($<$ 1h), many were solved using only a few minutes of human attention. While most of the examples only required maximizing a single loop counter, index, or pointer in combination with some variants of string comparisons, some of the cases are more interesting. In particular, NRFIN_00004, NRFIN_00041, and CROMU_00020 required multiple steps to solve. To showcase these more complex results, we now discuss the techniques used in three case studies.

TABLE VIII: Solving CGC Challenges. We give both the number of lines of code (LOC) that were used for IJON annotations, and an estimate for the human effort that went into producing the solution. Solutions annotated with * are discussed in more detail. † indicates that the `strcmp` could be solved using a properly chosen dictionary.

| Target | LOC | Effort | Comment |
|---|---|---|---|
| CROMU_00011 | 2 | $<$ 1h | strcmp, maximize index |
| NRFIN_00030 | 1 | $<$ 1h | maximize index |
| NRFIN_00004 | 5 | $<$ 5h | strcmp*, maximize index |
| NRFIN_00076 | 1† | $<$ 1h | strcmp |
| NRFIN_00041 | 4 | $<$ 1h | checksum* |
| CROMU_00020 | 3 | $<$ 5h | challenge response* |
| NRFIN_00005 | 1† | $<$ 1h | strcmp |
| NRFIN_00012 | 2 | $<$ 5h | strcmp |
| NRFIN_00038 | 1† | $<$ 1h | strcmp |
| NRFIN_00049 | 1† | $<$ 1h | strcmp |

### A. NRFIN_00004 (HeartThrob)

This program uses a fully unrolled prefix tree (*trie*) to perform string comparison. The function that performs the checking has roughly 30k different branches, and a correspondingly large number of possible paths. The fuzzer explores all inputs equally, filling the bitmap and producing useless inputs. After diagnosing the problem, it took us less than 20 minutes to build a small script that extracts the relevant strings from the trie and to disable coverage feedback within the function. Using these strings, we obtained the coverage that was needed. Lastly, we had to use the maximize primitive IJON-MAX to trigger an OOB crash.

### B. NRFIN_00041 (AIS-Lite)

This program uses an encoded checksum to guard the bug. After manually removing the checksum check, the bug was found very quickly. To obtain a valid input without understanding the format or the checksum, we used the following trick: we used an `IJON_CMP` and annotation on the checksum check to produce a valid input (while still using the patched target). AFL's crash exploration mode ensured that the fuzzer did not remove the cause of the crash while fixing the checksum. The fixed input triggers the crash in the unmodified binary. This approach closely mirrors the approach used by T-FUZZ [43] or REDQUEEN [7]. However, instead of leveraging symbolic execution or colorization, we used the fuzzer itself to provide the fixed input.

### C. CROMU_00020 (Estadio)

Similar to the previous case study, we used a "patch check and fix the crashing input afterwards" approach to solve this target. However, instead of a checksum, a series of challenge-response messages were required to trigger the bug.

## REFERENCES


[1] Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/. Accessed: February 12, 2020.

[2] DARPA Challenge Binaries on Linux, OS X, and Windows. https://github.com/trailofbits/cb-multios. Accessed: February 12, 2020.

[3] DARPA Cyber Grand Challenge binaries. https://github.com/CyberGrandChallenge. Accessed: February 12, 2020.

[4] Project Triforce: Run AFL on Everything! https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/. Accessed: February 12, 2020.

[5] Security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz. Accessed: February 12, 2020.

[6] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[7] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[8] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[9] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. Grimoire: Synthesizing structure while fuzzing. In *USENIX Security Symposium*, 2019.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[12] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[13] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[14] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, 2018.

[15] David R Cok and Scott C Johnson. Speedy: An eclipse-based ide for invariant inference. *arXiv preprint arXiv:1404.6605*, 2014.

[16] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-wide security testing of real-world embedded systems software. In *usenix-security*, 2018.

[17] Jerry Alan Fails and Dan R. Olsen, Jr. Interactive machine learning. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, 2003.

[18] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. Weizz: Automatic grey-box fuzzing for structured binary formats. *arXiv preprint arXiv:1911.00621*, 2019.

[19] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security and Privacy*, 2018.

[20] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[22] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Symposium on Network and Distributed System Security (NDSS)*, 2008.

[23] Google. Open sourcing ClusterFuzz. `https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html`. Accessed: February 12, 2020.

[24] Deepak Gopinath, Siddarth Jain, and Brenna D Argall. Human-in-the-loop optimization of shared autonomy in assistive robotics. *IEEE Robotics and Automation Letters*, 2016.

[25] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. AntiFuzz: Impeding Fuzzing Audits of Binary Executables. In *USENIX Security Symposium*, 2019.

[26] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, 2013.

[27] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[28] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. Vdf: Targeted evolutionary fuzz testing of virtual devices. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2017.

[29] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[30] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. Fuzzification: Anti-fuzzing techniques. In *USENIX Security Symposium*, 2019.

[31] S. Lem. *The Star Diaries: Further Reminiscences of Ijon Tichy*. HMH Books, 2012.

[32] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state Based Binary Fuzzing. In *Joint Meeting on Foundations of Software Engineering*, 2017.

[33] LLVM. Libfuzzer. `https://llvm.org/docs/LibFuzzer.html`. Accessed: February 12, 2020.

[34] Heather Logas, Jim Whitehead, Michael Mateas, Richard Vallejos, Lauren Scott, Daniel G Shapiro, John Murray, Kate Compton, Joseph C Osborn, Orlando Salvatore, et al. Software verification games: Designing Xylem, The Code of Plants. In *International Conference on the Foundations of Digital Games (FDG)*, 2014.

[35] Check Point Software Technologies LTD. Symbolic execution in vuln research. `https://research.checkpoint.com/50-adobe-cves-in-50-days/`. Accessed: February 12, 2020.

[36] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[37] David Molnar, Xue Cong Li, and David Wagner. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *USENIX Security Symposium*, 2009.

[38] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach. In *ACM Symposium On Applied Computing (SAC)*, 2018.

[39] Martin Ouimet and Kristina Lundqvist. Formal software verification: Model checking and theorem proving. Technical Report ESL-TIK-00213, Mälardalen University, 2007.

[40] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Validity fuzzing and parametric generators for effective random testing. In *International Conference on Software Engineering (ICSE)*, 2019.

[41] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[42] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[43] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy*, 2018.

[44] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *arXiv preprint arXiv:1811.09447*, 2018.

[45] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[46] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan M Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium*, 2014.

[47] David Romero, Peter Bernus, Ovidiu Noran, Johan Stahre, and Åsa Fast-Berglund. The operator 4.0: human cyber-physical systems & adaptive automation towards human-automation symbiosis work systems. In *IFIP International Conference on Advances in Production Management Systems*, 2016.

[48] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson Education Limited, 2016.

[49] Gunar Schirner, Deniz Erdogmus, Kaushik Chowdhury, and Taskin Padir. The future of human-in-the-loop cyber-physical systems. *IEEE Computer*, 2013.

[50] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *USENIX Security Symposium*, 2017.

[51] Stacey D Scott, Neal Lesh, and Gunnar W Klau. Investigating human-computer optimization. In *ACM CHI Conference on Human Factors in Computing Systems (CHI)*, 2002.

[52] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[53] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[54] Dimitri Vyokov. Syzkaller. `https://github.com/google/syzkaller`. Accessed: February 12, 2020.

[55] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[56] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[57] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Hongxu Chen, Minhui Xue, Bo Li, Yang Liu, Jianjun Zhao, Jianxiong Yin, and Simon See. Coverage-guided fuzzing for deep neural networks. *arXiv preprint arXiv:1809.01266*, 2018.

[58] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[59] Fisher Yu, Ari Seff, Yinda Zhang, Shuran Song, Thomas Funkhouser, and Jianxiong Xiao. Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop. *arXiv preprint arXiv:1506.03365*, 2015.

[60] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[61] Michał Zalewski. american fuzzy lop. `http://lcamtuf.coredump.cx/afl/`. Accessed: February 12, 2020.

[62] Michal Zalewski. Symbolic execution in vuln research. `https://lcamtuf.blogspot.com/2015/02/symbolic-execution-in-vuln-research.html`. Accessed: February 12, 2020.

[63] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.
[64] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *USENIX Security Symposium*, 2019.