The Use of Likely Invariants as Feedback for Fuzzers

Andrea Fioraldi
EURECOM
fioraldi@eurecom.fr

Daniele Cono D'Elia
Sapienza University of Rome
delia@diag.uniroma1.it

Davide Balzarotti
EURECOM
balzarot@eurecom.fr

Abstract

While fuzz testing proved to be a very effective technique to find software bugs, open challenges still exist. One of the its main limitations is the fact that popular coverage-guided designs are optimized to reach different parts of the program under test, but struggle when reachability alone is insufficient to trigger a vulnerability. In reality, many bugs require a specific program state that involve not only the control flow, but also the values of some of the program variables. Unfortunately, alternative exploration strategies that have been proposed in the past to capture the program state are of little help in practice, as they immediately result in a state explosion.

In this paper, we propose a new feedback mechanism that augments code coverage by taking into account the usual values and relationships among program variables. For this purpose, we learn *likely invariants* over variables at the basic-block level, and partition the program state space accordingly. Our feedback can distinguish when an input violates one or more invariants and reward it, thus refining the program state approximation that code coverage normally offers.

We implemented our technique in a prototype called InvsCov, developed on top of LLVM and AFL++. Our experiments show that our approach can find more, and different, bugs with respect to fuzzers that use a pure code-coverage feedback. Furthermore, they led to the discovery of two vulnerabilities in a library tested daily on OSS-Fuzz, and still present at the time in its latest version.

1 Introduction

Thanks to its success in discovering software bugs, *Fuzz Testing* (or *fuzzing*) has rapidly become one of the most popular forms of security testing. While its original goal was simply to randomly generate unexpected or invalid inputs, today's fuzzers always rely on some form of heuristics to *guide* their exploration. The most popular of these strategies is, by far, *Coverage-Guided Fuzzing* (CGF), in which the fuzzer selects inputs that try to increase some coverage metric computed over program code—typically, the number of unique edges in the control flow graph. Consequently, a large body of research has focused on overcoming the limitations of coverage-guided fuzzers, for instance by proposing techniques to solve complex path constraints [82] [66] [77] [69] [5], by reducing the large number of invalid testcases generated by random mutations [65] [59] [3] [6] [28], or by focusing the exploration on more 'promising' parts of the program [58] [57] [9].

While these improvements have considerably decreased the time required to visit different parts of the target application, it is important to understand that code coverage alone is a necessary but not sufficient condition to discover bugs. In fact, a bug is triggered only when i.) program execution reaches a given instruction, and ii.) the state of the application satisfies certain conditions. In rare cases, there are no conditions on the state, as it is the case for most of the bugs in the Lava-M [22] dataset—which were artificially created to be triggered by simply reaching a certain point in the target applications [38].

On the one hand, this aspect is very important because the use of code coverage to reward the exploration results in the fact that fuzzers do not have any incentives to explore more states for an already observed set of control-flow facts (e.g., branches and their frequencies). Thus, it is considerably harder for existing tools to detect bugs that involve complex constraints over the program state. On the other hand, the simple solution of rewarding fuzzers for exploring new states (state coverage) is also a poor strategy, which often *decreases* the bug detection rate. This is due to the fact that, for non-trivial applications, the number of possible program states is often infinite.

Therefore, special techniques are needed to reduce the program state into something more manageable to explore during testing, while still preserving the fuzzer's ability to trigger potential bugs. To date, few works have tried to find such compromise. For instance, some fuzzers approximate the program state by using more sensitive feedbacks, like code coverage enriched with call stack information, or even with values loaded and stored from memory. This second approach, as shown in [79], better approximates the program state coverage by taking into account not only the control flow but also the values in the program state, but is less efficient

than others in finding bugs as it incurs into the state explosion problem mentioned above.

To capture richer state information while avoiding the state explosion problem, researchers have also looked at human-assisted solutions. For instance, FUZZFACTORY [60] lets the developers define their domain-specific objectives and then adds waypoints that reward a fuzzer when a generated testcase makes progress towards those objectives (e.g., when more bits are identical among two comparison operands).

At the time of writing, the most successful approximation of the program state coverage is achieved by targeting only certain program points selected by a human expert, as recently proposed in [4]. In the work, portions of the state space are manually annotated and the feedback function is modified to explore such space more thoroughly. We believe that the automation of this process may be a crucial topic in future research in this field.

Our Approach. In this paper, we propose a new feedback for Fuzz Testing that takes into account, alongside code coverage, also some interesting portions of the program states in a fully automated manner and without incurring state explosion.

The key idea is to augment edge coverage—the most widely-adopted and successful code coverage metric used by fuzzers—with information about local divergences from 'usual' variable values. To this end, we mine *likely invariants* on program variables by executing an input corpus (such as the queue extracted from a previous CGF campaign) and learning constraints on the values and relationships of those variables over all the observed executions. It is important to note that execution-based invariant mining produces constraints that do not necessarily model properties of the program, but rather local characteristics of the analyzed input corpus [25]: hence, constraints may be violated under different inputs.

Our intuition is that these local properties represent an interesting abstraction of the program state. We thus define a new feedback function that treats an edge differently when the incoming basic block sees one or more variable values that violate a likely invariant. This approach increases the sensitivity of a standard CGF system, rewarding the exploration of program states that code coverage alone would not be able to distinguish.

We develop a set of heuristics to produce and refine invariants, and techniques to effectively instrument programs with a low-performance overhead—a very important metric in fuzzing. We implement them into a prototype called INVSCOV on top of LLVM [43] and the AFL++ [30] fuzzer.

Our experiments, conducted over a set of programs frequently tested by other fuzzers, suggest that our feedback, by succinctly taking into account information about usual program state in addition to control flows, can uncover both more and different bugs than classic CGF approaches.

Contributions. In summary, the main contributions of this paper are:

- A new feedback that combines control flows with an abstraction of the program state from mined invariants;
- A prototype implementation of our approach based on LLVM and AFL++ called INVSCOV;
- An evaluation of the effectiveness of our approach against classic and context-sensitive edge coverage.

We share the INVSCOV prototype as Free and Open Source Software at https://github.com/eurecom-s3/invscov.

2 Background

This section covers key concepts of invariant mining and Fuzz Testing techniques that are pivotal to our proposal.

2.1 Program Properties and Invariants

Property-based testing is a software testing methodology in which some form of specification of the program's properties drives the testing process. Such specification simultaneously defines what behaviors are valid and serves as basis for generating testcases [27].

The correctness oracle can be embedded in the target program itself in the form of a set of assertions that check the validity of each *invariant*, i.e., a property that according to the specification must always hold at that program point [26]. Testcases can then be generated by aiming at violating the invariant assertions.

Since delegating the identification of program properties to the developers can be a daunting prospect, automation has been the subject of a large body of previous works in the field. Automated invariant learning is also a widely explored topic in other areas, for instance for memory error detection [37] (we will discuss some of these alternative lines of work in more details in §6.1).

Invariants can be discovered by conducting static code analysis: for instance, RCORE [32] builds on abstract interpretation [14] and monitors invariants at run-time to detect program state corruption from memory errors. Generally, such invariants are sound and incur limited false positives, yet the inherent over-approximation of static analysis may generate invariants too coarse to discriminate program states in an effective manner for high-level analysis.

Therefore, a more precise way to discover invariants, which also produces them in greater quantity, consists of inspecting the program state at run-time. For this reason, approaches like [35], [24], and [61] build on information gathered during the execution, in a dynamic fashion. The downside of dynamic approaches is that, unlike static ones, they produce *likely invariants*, i.e., invariants that hold for the analyzed traces but may not hold for all inputs. Hence, they may result in false positives when the learned invariants capture only local properties of the observed executions.

In this work we build upon this well-known coverage problem [26] and turn it into an advantage for driving a fuzzer. We do that by starting from a corpus of testcases that—as it

is the case with real applications—cannot be representative of all program states, then we modify a fuzzer to make it more sensitive to behaviors that diverge from the likely invariants obtained from the initial corpus. In this case, the fact that the learned invariants capture properties of the observed executions instead of properties of the program itself is the key intuition we use to generate a more diverse set of input values.

2.2 Fuzz Testing

Fuzz Testing, or *fuzzing*, is a family of software testing techniques first proposed in the '80s. Recently, fuzzing techniques saw significant improvements in their effectiveness, and contributed to the discovery of many security vulnerabilities [62] [50]. Nonetheless, the key idea behind Fuzz Testing research remained simple: repeatedly execute the program under test by using randomly generated inputs, usually chosen to be either unexpected or invalid. Fuzzing tools monitor a program for failures, such as invalid memory accesses or out-of-memory crashes, and report to the user the inputs that triggered such behaviors.

The most naive embodiment of fuzzing just provides random inputs to the program under test without any knowledge about its characteristics (e.g., input format) or the program execution. This approach, albeit still effective in testing legacy code [54], has obvious limitations. Therefore, many different solutions have been proposed over the past decades [50] [62] to increase the effectiveness in bug finding far beyond naive fuzzing. We can group these techniques according to the following three criteria: 1) the amount of information they require to know from the program, 2) the technique they use to generate new testcases, and 3) the feedback they use to guide the exploration.

According to the first criterion, we can distinguish three main categories of fuzzers:

- *White-box* fuzzers, which build a full picture of the program using program analyses. Concolic executors like SAGE [33] and SYMCC [66] belong to this category, as they collect a model of the program in terms of logic constraints during the execution. The cost of such white-box analyses, however, may often be untenable [62];
- *Black-box* fuzzers, which blindly generate random inputs for testing. They can access knowledge about the input format, but generate inputs regardless of how the program implementation looks like [80] [52];
- *Grey-box* fuzzers, which fall halfway between the two previous categories. They access limited information provided by a lightweight instrumentation applied to the program under test, blending the program analysis and testing stages [62]. An example of such information is the *code coverage* extracted from a testcase by systems like AFL [83] and LIBFUZZER [46].

According to our second criterion, we can distinguish instead fuzzers based on their input generation methodology. The two most commonly used approaches in this respect are *generational* and *mutational* fuzzers. A generational fuzzer creates new testcases from scratch, either randomly or by relying on some form of format specification—like a grammar [40] or a domain specific language [23]. Mutational fuzzers instead derive new testcases from a set of prior testcases by mutation; the mutations can be generic [83], target-specific [78], or driven by a user-supplied [3] [65] or inferred [6] [28] format specification.

Finally, by using our third and last criterion, fuzzers can be divided according to the information they use to drive their exploration, which we call *Feedback*. A popular and very effective technique is coverage-guided fuzzing, which uses code coverage as feedback to drive the testcase generation. Previous studies have shown that coverage-based fuzzers are often one order of magnitude more effective at discovering bugs [19]. As also other forms of feedback are possible, we will refer more in general to this fuzzing design as *Feedback-Driven Fuzz Testing*.

2.2.1 Feedback-Driven Fuzz Testing

In short, when a CGF solution generates a testcase that triggers a previously unexplored portion of the program, it deems the testcase as *interesting* and adds it to a *queue* of inputs (dubbed *seeds*) maintained for further processing. By combining this technique with a mutational approach, we obtain an evolutionary algorithm driven by code exploration.

Code coverage can be measured in different ways, for instance by considering basic blocks alone or by including entire calling contexts [79]. By far, the most popular criterion used for coverage-guided fuzzers is *edge coverage*, which maximizes the number of edges visited in the control flow graph (CFG) of program functions. Fuzzers like AFL [83] extend pure edge coverage by also including a hit count for edges (i.e., how many times a testcase exercises them) to better approximate the program state. Recently, ANKOU developed this idea further by adding coverage-equivalent testcases to the queue depending on the results of an online principal component analysis for hit count differences between executions.

As we anticipated in §2.2, other metrics are possible for driving fuzzer evolution. FUZZFACTORY [60] recently studied several alternatives, such as the fact that the size of memory allocations can be a useful feedback to expose out-of-memory bugs, while the number of identical bits in the operands of a comparison instruction [45] can help in circumventing fuzzing roadblocks (§6.2). In short, all these feedback techniques act as shortcuts to domain-specific testing goals for which code coverage is not an adequate description.

A more general approach would be to consider, alongside control flow decisions, also data flow information regarding the program state. The most naive embodiment of this feedback—and to the best of our knowledge also the sole to date—is the 'memory' feedback, where every newly observed data values from memory load and store operations are

```
1  int wavlike_msadpcm_init (SF_PRIVATE
       *psf, int blockalign, int samplesperblock)
2  {  MSADPCM_PRIVATE  *pms ;
3    unsigned int pmssize ;
4    // Likely Invariants:
5    //  - blockalign ∈ { 0, 2, 256 }
6    //  - blockalign < samplesperblock
7    ...
8    pmssize = sizeof (MSADPCM_PRIVATE) + blockalign
       + 3 * psf->sf.channels * samplesperblock ;
9    ...
10   pms->samples   = pms->dummydata ; // array in pms
11   pms->block     = (unsigned char*) (pms->dummydata
       + psf->sf.channels * samplesperblock) ;
12   pms->channels  = psf->sf.channels ;
13   pms->blocksize  = blockalign ;
14    ...
15 }
```

Listing 1: Excerpt of wavlike_msadpcm_init() initialization code.

```
1  static int msadpcm_decode_block
       (SF_PRIVATE *psf, MSADPCM_PRIVATE *pms)
2  {
3    ...
4    sampleindx = 2 * pms->channels ;
5    // Likely Invariants:
6    //   - pms->blocksize == 256
7    while (blockindx < pms->blocksize)
8    {    bytecode = pms->block [blockindx++] ;
9         pms->samples [sampleindx++]
         = (bytecode >> 4) & 0x0F ; // heap overflow bug
10
         pms->samples [sampleindx++] = bytecode & 0x0F ;
11       } ;
12   ...
13 }
```

Listing 2: Vulnerable code found in msadpcm_decode_block().

considered as novelty factor for the fuzzer. Unfortunately, this solution easily leads to state explosion [79].

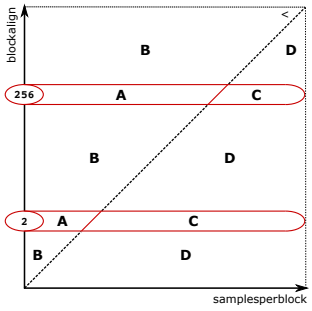
3 Methodology

In this section, we present the intuition behind our approach by using an example of a real-world vulnerability we discovered during our experiments. The vulnerability is a heap overflow in the WAV file format parsing of `libsndfile`, a popular library to operate on audio files. Listings 1 and 2 show the affected code. Specifically, the vulnerability is located in the `msadpcm_decode_block` function of file `ms_adpcm.c`, reported here at line 9 in Listing 2.

For our purpose, it is interesting to note that all the coverage-guided fuzzers we used in our experiments (§5) were able to reach the vulnerable point in the code without, however, triggering the bug. Despite the fact that the vulnerable code is 'easy-to-reach' and that `libsndfile` is often used in fuzzing experiments (including the Google OSS-Fuzz project and recent research works such as [31] and [81]), the bug was still present when we ran our experiments.

This is likely due to the fact that to trigger the bug the loop should write outside the memory pointed by `pms->samples`, which references the C99 variable-size array field at the end of the `pms` structure. This only happens when the program is in a specific state, characterized by a small allocation size for the `pms` buffer (line 8 in Listing 1) and a `pms->blocksize` value (line 13 in Listing 1) sufficiently high to force the loop to write out of the bounds of the array.

However, none of these requirements can be extracted from code coverage, as there are no branches in the program that involve these thresholds. Instead, they both depend on two input-derived values: `blockalign` and `samplesperblock`. Hence, a CGF-based exploration may easily satisfy one of the requirements but, without recognizing this as progress in the program exploration, it would unlikely satisfy both at



Invariant	Condition
LI_1	blockalign $\in \{0,2,256\}$
LI_2	blockalign $<$ samplesperblock

Invariant	A	B	C	D
LI_1	✓	✗	✓	✗
LI_2	✗	✗	✓	✓

Figure 1: State partitioning for wavlike_msadpcm_init() induced by the two likely invariants LI_1, LI_2. The bug can be exercised only when in partition B (LI_1, LI_2 both violated).

the same time. In fact, any generated testcase satisfying either requirement would exercise an "intermediate" program state closer to the bug, but would not be seen as an interesting one to add to the queue for more mutations, because in the eyes of CGF it does not bring novel code coverage.

This example shows the challenge that modern fuzzers encounter when exploring the state of a program, even for code that does not entail difficult path conditions to be reached. State-of-the-art CGF systems can saturate in coverage while still missing bugs at program points touched in their operation. Also, they may fail to generate testcases to cover unseen program points whenever those are reachable only upon meeting conditions that do not depend on control flow alone.

3.1 Program State Partitions

The core idea of this paper is that we can divide the program space in different partitions at multiple points in the application code, by learning likely invariants from executing the program under test over an initial corpus of inputs.

To continue with our example, let us imagine that we can fuzz `libsndfile` for a certain amount of time, e.g., 24h, with a standard CGF system (we will discuss in §3.4 the effect of

different corpora on the extracted invariants). By investigating the values of the variables across all seeds saved by the fuzzer, we would identify two likely invariants for the init function and one for the vulnerable decoding loop. All invariants are included as comments in Listings 1 and 2.

It is important to understand that these invariants are descriptive of the limited number of states that were induced by the corpus generated by the fuzzer. In other words, each invariant expresses a condition over the state of the program that the fuzzer was unable to violate during the testing experiment. Therefore, our intuition is that we can use these invariants to divide the program state into a number of partitions, as depicted in Figure 1 for the init function.

In this case, we can see that the two invariants partition the space in four non-contiguous areas (*A* to *D* in the figure), all but the first *unvisited* by the fuzzer. This information allows us to provide feedback to the fuzzer to explore new abstract states without incurring into the classic state explosion problem.

Moreover, since these states can be reached only by violating the invariants we learned over previous executions of the fuzzer, our intuition is that they are likely to bring the program into seldom-explored corner cases—where vulnerabilities may lie undetected for a long time.

To capture this information, the approach presented in this paper augments the classic edge coverage feedback by using the violation of likely invariants learned over basic blocks. In an ideal world, we could learn exact invariants and transform them in terms of code coverage, allowing pure coverage-based fuzzers to receive feedback to progress towards these areas. However, as described in §2.1, current invariant mining techniques lead to both over or under approximations.

3.2 Using Invariants as Feedback

The common limitation of dynamic invariant detection is that the resulting invariants often capture local properties of the test suite more than static properties of the program.

However, for our purpose, this is exactly what we want. In fact, likely invariants that represent only local properties of the corpus are interesting because their violation would tip fuzzers about what value combinations in the program state are unusual, and ideally the home of bugs.

Therefore, we define our invariant-based feedback as a combination of edge coverage with the information about which likely invariants are violated in the source basic block. To inform the fuzzer about the progress towards interesting states, we then tweak the classic novelty search algorithm adopted by most coverage-based systems. In particular, for each CGF-instrumented control flow graph edge, we make it generate a different value for the novelty search for each unique combination of violated invariants. As we will detail in Section 4.2, we track invariants individually and reward them independently at each basic block: this choice brings an unambiguous, implicit encoding of program state partitions.

The invariants ability to partition the program state space without incurring state explosion is also one of the key insights of our approach. At each basic block N invariants can partition the state locally just like N non-parallel lines can divide a plane into $N*(N+1)/2+1$ regions. In practice, since each basic block typically manipulates only few variables, N is usually a very low value (statistics in Appendix A).

Back to our example, for the `wavlike_msadpcm_init` function we have two variables involved in the learned invariants: `blockalign` and `samplesperblock`. The partition that triggers the vulnerability is B—the one that sees both invariants violated. Our fuzzer found the bug for a value assignment `{blockalign = 1280, samplesperblock = 8}`.

With the enriched sensitivity from our invariant-based feedback, the fuzzer can violate each invariant separately, save such testcases for partitions A and D, and for instance splice the two testcases to generate one that brings the state to B. More in general, our approach can generate inputs that violate multiple invariants by either combining or mutating previous seeds—each violating one or more distinct invariants.

As for the likely invariant involving `pms->blocksize` in the buggy function (Listing 2), we observe that violating it is not a sufficient condition to trigger the bug. The field is assigned equal to `blockalign` in Listing 1, but also `samplesperblock` has to contribute to expose the bug.

3.3 Pruning the Generated Checks

With our example, we showed how we can use invariants to partition the program state and how we can then provide this information as feedback to drive the fuzzer's exploration.

However, not all invariants are equally useful: while having more invariants does not affect our methodology (i.e., we do not lose sensitivity by exposing more partitions), the extra states they generate can pollute our feedback and the additional instrumentation can impact the run-time overhead.

Therefore, we designed three classes of pruning rules to remove invariants that would be fruitless to check either in light of other available information or because of the nature of their constituents.

1. The first class of invariants we discard are those that are **impossible** to violate. For instance, our likely-invariant mining system would often learn that unsigned integer variables are always greater than or equal to zero—which is not a very useful condition to drive a fuzzer. To identify these and alike cases, we perform a *Value Range Analysis* [36] for each function of the program under test. Arguments and global storage are initially seen unconstrained, and the analysis produces bounds for function variables that hold for any execution. Using range information, we instruct our miner to never generate likely invariants that are logically weaker than the ones found statically. Since these invariants cannot be violated, we can save the instrumentation cost required to monitor them.

2. The second class of fruitless invariants are those that combine **unrelated variables**. To remove these relationships, we compute *Comparability Sets* for each function of the program under test: each variable belongs to only one such set, and invariants combining variables across different sets are discarded. We initially create a separate set for each variable, then use a unification-based policy by iterating over function instructions and merging the sets of two variables whenever those occur as operands for the same statement. Eventually, a comparability set contains variables that take part in related computations. Few exceptions apply: for instance, in an array pointer computation we do not merge the sets of the base and the index elements as they are not directly related.

3. Whenever different invariants have **overlapping conditions**, it is possible to optimize their run-time verifications by reusing previously computed values. In particular, we target pairs of likely invariants that share the same conditions on some of their variables. If the two invariants concern two program points p and p' where p' can execute only after p, we can use a standard flow-sensitive analysis to determine whether between p and p' there are no intervening re-definitions for any of the involved variable. In that case, we simply propagate the value computed at p and save the computation cost at p'.

The output of the value-range analysis and the comparability sets are computed beforehand and passed to the invariant miner, which takes them into account when generating the invariants. Overlapping conditions are instead dealt with when producing the program—augmented with code for checking invariants—that will undergo the testing process.

3.4 Corpus Selection

For our entire solution to work, we need to be able to learn likely invariants from a large number of executions of the program under test. Therefore, like for many other evolutionary fuzzing techniques, the choice of the initial corpus of inputs is critical.

An unwise choice can generate invariants that do not describe with sufficient generality the shape of the variables in the program state. For instance, it is a common practice in fuzzing to download many files of a given file format when testing a parser, but almost all those files are valid files. If we learn likely invariants from the program executions of such a corpus, we will bias our invariants on the validity of the file format and, in some cases, this can be a mistake because we might miss interesting partitions of the program state related to invalid inputs.

As we want to address the problem of finding bugs even when the fuzzer saturates in coverage [34], a natural choice is to use as corpus the queue of a coverage-guided fuzzer taken as soon as that fuzzer shows signs of slowing down in reaching new coverage points. A violation of an invariant learned over such corpus will lead to novel feedback for the fuzzer and desaturate the search.

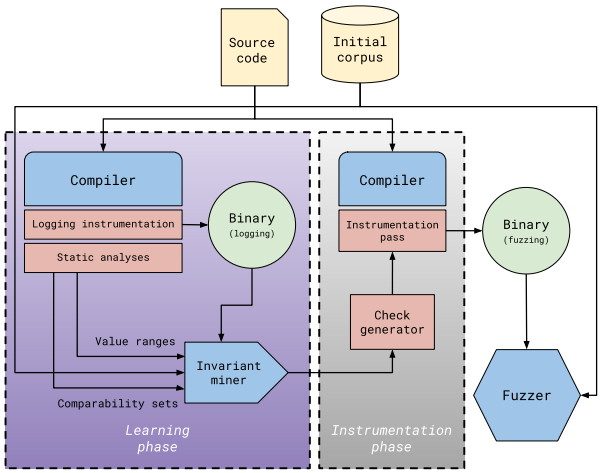


Figure 2: High-level workflow of invariant-based fuzzing.

To confirm our intuition we downloaded a dataset of valid files for the programs we tested in §5 and mined likely invariants by using such testcases. We then compared the invariants extracted from these initial seeds with those obtained using the queue after a 24h run of a coverage-based fuzzer initially supplied with the same seeds. In our experiments, we observed that the invariants extracted only by using the valid files led to the discovery of 20% fewer unique bugs than with the invariants extracted from an initial run of a fuzzer.

4 Implementation

In the previous section, we introduced the motivation and the key ideas behind our approach. However, we intentionally avoided discussing two important aspects of our solution: i.) how we define the *state* we want to capture in our invariants, and ii.) how we perform the instrumentation of the program under test to collect the information required by our technique.

Our approach can be implemented in different ways, for instance by instrumenting the target source code, or by performing binary-level instrumentation via static rewriting [20] or dynamic translation [17]. While each approach has its own pros and cons, for our experiments we opted for a compiler-based implementation of our invariant-based fuzzing using LLVM [43] and the DAIKON [25] likely-invariants system.

Our prototype is written in C++ and re-uses the fast intra-procedural integer range analysis of Pereira et al. [68] for LLVM, which takes an asymptotically linear time to complete. Figure 2 provides a high-level view of the complete architecture. We implemented two custom compile-time transformation phases (consisting of roughly 5 KLOC) for LLVM:

1. *Learning phase*, where we emit logging instrumentation for program state variables to feed the invariant miner;

2. *Instrumentation phase*, where we augment the code of

the program under test to evaluate the likely invariants in a form directly suitable for coverage-guided fuzzers.

In short, during the *Learning phase* we record all the information about the program state required for invariant mining. We achieve this by running an augmented version of the program under test over a corpus of inputs, which can be obtained in several ways (§3.4; in the experiments described in §5 we use the seeds generated from a 24h coverage-guided fuzzing session). For invariant mining we use the DAIKON dynamic invariant detector, one of the most used dynamic miners: first presented in 2007, DAIKON is still under active development.

At each instrumentation place, invariant mining faces a cubic time complexity in the number of constituents (i.e., program variables) [26]. However, since our technique is applied at the level of basic blocks, the number of variables is practically a small constant, and the total computation cost for invariant mining becomes linear in the number of basic blocks in the program.

During the *Instrumentation phase*, we then encode likely-invariant information in program functions to expose them to coverage-guided fuzzers. Our transformed programs can execute out of the box on any AFL-based fuzzer but, as we elaborate in more details in §4.2, we foresee minimal adaptations to support coverage tracking schemes from other fuzzer families.

4.1 State Invariants Learning

In order to learn the likely invariants, we need to observe the values of the program state during the execution of the program over the initial corpus of inputs. To achieve that, we compile a dedicated version of the program under test that includes additional instrumentation to collect such values at run-time.

Since our prototype is implemented on top of the *Intermediate Representation* (IR) of LLVM, we can easily expose the state of the program at the level of each basic block. Also, the IR allows us to avoid issues with uninitialized values that affect tracing complex data types at the source code level [1]. For instance, a structure may contain a pointer, and to extract the present pointed value for tracing purposes its address must be valid. The original Kvasir front-end of DAIKON uses expensive dynamic binary instrumentation [17] to read variables and inspect memory. However, by working at the IR level, we can just wait until the address appears in a virtual register as the result of a load operation and use it for tracing.

Another advantage of using an Intermediate Representation is that, in an IR, instructions are typically expressed in a Single Static Assignment (SSA) form [70]. SSA entails that each variable can only be assigned once, and each use must be reached by a (unique) prior definition.

For simplicity, in our implementation we ignore floating-point instructions and model the program state by looking at SSA variables holding integer values. For local variables, since multiple SSA variables exist in the IR for a single source-level variable[1], we restrict our analysis to those SSA variables that can be directly connected to a source-level variable, by using debug metadata from the LLVM front-end.

When a program instead accesses non-local storage or a field of a non-primitive type, LLVM introduces an SSA variable as result of a load operation for the current contents. By instrumenting such IR variables, our invariant mining extends also to global variables, heap storage, and fields of structs.

Moreover, since our goal is not just to model the state of an application, but to improve the effectiveness of a security-oriented testing technique, we focus our analysis on those variables that can have security-related consequences, according to the following three rules:

- The variable is part of a `GetElementPtr` instruction[2] for pointer computation unless only constant indexes are involved;
- The variable value is loaded from or stored to memory by using a `Load` or `Store` instruction;
- The variable represents the return value of a function.

To collect the value of each variable we implemented an LLVM function pass that, alongside instrumenting the variables of interest with logging machinery, also dumps at compilation time the Comparability sets and the integer ranges [68] to support the pruning techniques described in §3.3.

The pass creates a JSON file for each code module to store information about program points and variables (type, comparability, and bounds). We then process and merge these intermediate files from all modules to produce the DAIKON declaration file[3], adding also comparability and range bound information for the sake of invariant pruning (Section 3.3). We instruct DAIKON to run the instrumented program over each input in the corpus and retrieve the values logged for its variables. We mine our invariants by using the on-demand mode of DAIKON, which learns incrementally from each execution.

4.2 Program Instrumentation

In the second phase of our approach, we embed the likely invariants obtained from the Learning phase in the program under test and add the required AFL instrumentation to drive the fuzzer. For this, we turn each invariant into a C function that we compile to LLVM IR and invoke from the program point of interest. The function takes as arguments the IR values that are part of the invariant and evaluates them, returning a unique identifier when the invariant is violated, and zero otherwise. Listing 3 provides an example of such functions, generated for an invariant with identifier `123` that checks whether `var0 > 1`.

To expose the violation of invariants as if there were a code coverage change, we modify few lines that are part of the

[1] Special ϕ-*functions* regulate the currently visible assignment when it depends on the CFG basic blocks the program traversed.

[2] https://llvm.org/doxygen/classllvm_1_1GetElementPtrInst.html

[3] https://plse.cs.washington.edu/daikon/download/doc/developer/File-formats.html#Declarations

```
unsigned __daikon_constr_123(int var0) {
  if (!(var0 > 1))
    return 123 << 1;
  return 0;
}
```

Listing 3: Example of generated C code from an invariant.

```
// Original AFL edge-coverage code
__afl_area_ptr[cur_loc ^ prev_loc]++;
prev_loc = cur_loc >> 1;

// Extended to capture violations of invariants
__afl_area_ptr[cur_loc ^ prev_loc]++;
prev_loc = cur_loc >> 1;
prev_loc ^= __daikon_constr_123(var0);
prev_loc ^= __daikon_constr_321(var2, var3);
```

Listing 4:
Classic and Extended AFL instrumentation for edge coverage.

classic AFL instrumentation, as depicted in Listing 4. In the original code, `cur_loc` represents the identifier assigned to the current block, and `prev_loc` is the right-shifted-by-one value of the previous block identifier. An edge coverage event is reported by XOR-ing these two variables and by incrementing the corresponding entry in the `__afl_area_ptr` coverage map. In this way, the code can also capture the number of times that the edge is executed modulo 256 (map values are 8-bit unsigned integers).

To include the information about the violated invariants into the AFL feedback, we encode the identifiers of the violated invariants into `prev_loc` by using the XOR operation. This allows each edge to also capture which invariants were violated in the source basic block. Listing 4 shows how we augment edge coverage with the combination of the outcome of the functions that check the invariants with identifiers 123 and 321. Note that zero is the identity element for XOR, so edge coverage is unaffected when an invariant is not violated (i.e., the invariant's function returns zero).

We insert our instrumentation by using an LLVM Function pass. During this phase, we also apply the optimization to remove overlapping conditions, as described in §3.3, by identifying those invariant evaluations in different blocks that perform the same checks on the same values. To minimize their number, and therefore avoid redundant instrumentation that could slow down the execution, we build the dominator tree [67] for each function of the target program and emit the check only at the top-level block in such tree that strictly dominates all the other blocks in which the same invariant appears. Thanks to the SSA form, the value returned for the check is guaranteed to be visible at its dominated blocks, and therefore we can avoid re-executing the evaluation function.

5 Evaluation

In our experiments we tackle the following research questions:

- **RQ1.** Are our invariant pruning heuristics effective in reducing the number of generated checks?
- **RQ2.** Does our new feedback incur state explosion?
- **RQ3.** Can our feedback lead a fuzzer to effectively exploring more program states than code coverage?
- **RQ4.** Can our feedback uncover more, or just different, bugs than code coverage?
- **RQ5.** What run-time overhead does our feedback introduce?

In order to answer these questions, we selected 8 real-world target programs as subjects for our experiments. We opted for programs that work on distinct file types and follow different strategies in the implementation of the parsing stage. In more detail, `cappt` and `xls2csv` look up tokens using large switch constructs, `jasper` works on a chunk-based format, `sndfile-info` is stream-oriented, `pcre2` uses lookup tables, `gm` combines different strategies, `exiv2` is chunk-based and uses C++ objects to represent chunks, and `bison` is an LR parser. The versions we selected are known to contain bugs as they are widely used in past works (e.g., [31] [51] [69]) to test fuzzers. For a rigorous evaluation we also manually deduplicate crashes when assessing bug finding capabilities [42].

Note that popular benchmarks like LAVA-M [22] are not suitable for evaluating our approach, as the bugs they contain depend exclusively on code reachability guarded by magic-value (§6.2) comparisons [38]. We also opted not to use the recent and appealing MAGMA [38] benchmarks, as their hardwired logging primitives (used to check for ground truth) split basic blocks and thus conflict with the granularity of our invariant construction and instrumentation.

To enable reproduction of our results, Table 1 lists the programs we used in our experiments, their software package and version, their lines of code, the command line used to test each program, and the sanitizers [76] enabled at compilation time. We applied both AddressSanitizer (ASAN) and UndefinedBehaviourSanitizer (UBSAN) compile-time instrumentation. However, we had to disable UBSAN for two applications as it introduced unwanted side-effects that made them crash even with the simplest test inputs.

Experimental Setup

We ran all experiments on a x86_64 machine equipped with an Intel® Xeon® Platinum 8260 CPU with a clock of 2.40 GHz. We used AFL++ version 2.65d as reference fuzzer to study the benefits of our approach and draw comparisons with the many configurations AFL++ offers (e.g., alternative mutation and seed scheduling policies, and context-sensitivity).

We ran each experiment 5 times to reduce the impact of fuzzing randomness, and report the median value to aggregate the results. Each experiment had a 48h budget.

Starting from an initial collection of valid files, we ran AFL++ for 24h and collected its queue as a corpus, which we used both as corpus for learning the likely invariants and as initial seeds for all the fuzzers we evaluate in our experiments. The same configuration was used in [6] for incremental

Program	Package	KLOC	Command line	Sanitizers
`catppt`	CATDOC 0.95	7	`@@`	ASAN, UBSAN
`xls2csv`	CATDOC 0.95	7	`@@`	ASAN, UBSAN
`jasper`	Jasper 2.0.16	176	`-f @@ -t jp2 -T mif -F /dev/null`	ASAN
`sndfile-info`	libsndfile 1.0.28	79	`-cart -instrument -broadcast @@`	ASAN, UBSAN
`pcre2` (harness)	PCRE2 10.00	68		ASAN, UBSAN
`gm`	GraphicsMagick 1.3.31	251	`convert @@ /dev/null`	ASAN, UBSAN
`exiv2`	Exiv2 0.27.1	80	`@@`	ASAN, UBSAN
`bison`	Bison 3.3	100	`@@`	ASAN

Table 1: List of target programs used for the evaluation along with the corresponding package, the lines of C/C++ code, the command line used for the fuzzers, and the sanitizers used when compiling each program.

	Invariant pruning		
Program	None	Learning	All
`catppt`	137	137 (100%)	136 (99%)
`xls2csv`	453	400 (88%)	396 (87%)
`jasper`	11459	9144 (80%)	9144 (80%)
`sndfile-info`	3462	3013 (87%)	2996 (86%)
`pcre2`	4992	4803 (96%)	4497 (90%)
`gm`	16173	14362 (89%)	13278 (82%)
`exiv2`	6040	5534 (91%)	4943 (82%)
`bison`	9363	6263 (67%)	5983 (64%)
Total	52079	43556	41373
% (w.r.t. Unopt.)	100%	84%	79%

Table 2: Number of generated checks without any optimization, with optimizations for learning phase only, and with optimizations for learning & instrumentation phases.

fuzzing runs and allowed CGF fuzzers to approach saturation in our tests.

Throughout the rest of the section, we will denote with INVSCOV a fuzzer that uses our invariant-based instrumentation as feedback, with CODECOV a fuzzer that uses classic edge coverage as feedback, and with CTXCOV a fuzzer that augments edge coverage with context sensitivity.

5.1 RQ1: Invariant Pruning

To answer the first research question, we measured how the pruning rules introduced in §3.3 ultimately impact the number of tests for likely invariants that our system needs to insert into the program under test.

Table 2 reports the number of checks generated without any optimization enabled, with only those for the learning phase (comparability sets and removal of invariants impossible to violate) enabled, and with also the optimization applied at the instrumentation phase (overlapping conditions).

The optimizations from the learning phase reduce the amounts of checks by 14% on average. This resulted in an average of 1.4 likely invariants generated for each basic block that accesses one or more profiled variables (§4.1) in the LLVM IR. Upon adding the overlapping-conditions optimization from the instrumentation phase, the total number of invariants decreased by 21%. While the overall reduction may seem small, according to our experiments the smaller number of invariants to check at run-time resulted in a 10% net increase in the performance of the fuzzer.

5.2 RQ2: State Explosion

The number of testcases maintained in the fuzzer's queue can serve well the purpose of verifying whether our technique would result in an explosion on the number of states the fuzzer has to track. In fact, the number of stored seeds is representative of the interesting testcases generated and therefore of distinct portions explored in the state space that is visible to the fuzzer. The first two columns of Table 3 report the number of testcases in the fuzzer's queue after a 48h session. The growth due to the use of invariants is moderate, and only accounts for a 62% increase across all programs.

This is very important because an excessively large queue becomes unmanageable for a fuzzer. Wang et al. [79] studied queue sizes for two memory-based feedbacks (§2.2.1) and reported growth factors of 21x and 14x as geometric mean for the DARPA CGC benchmarks, and peaks of 196x and 512x. The authors also observed that the relative differences among most seeds were so small that they were very unlikely to lead to the discovery of new bugs. On the contrary, more moderate increases, such as ~8x over edge coverage for feedbacks focused on control flows (e.g., n-grams, context-sensitivity), resulted in a profitable end-to-end bug finding.

In most of our programs we measured a growth factor below 2x, except for `jasper`, for which it was roughly 3x, yet far behind the numbers that were reported to cause state explosion in previous studies.

5.3 RQ3: Program State Exploration

Since our main goal is to help the fuzzer to explore various program states that can lead to bugs, we now look at how our proposed approach explores the program behaviors that would be visible to a pure code coverage-based approach.

First of all, we study the (cumulative) edge coverage on

Program	Testcases		Edges		Violated Checks		Exec / Sec	
	InvsCov	CodeCov	InvsCov	CodeCov	InvsCov	CodeCov	InvsCov	CodeCov
catppt	213	119	404	404	40	5	112	101
xls2csv	1358	770	1013	1007	113	13	132	128
jasper	10831	3188	5452	5487	971	462	143	166
sndfile-info	1764	1297	8164	8074	558	214	151	152
pcre2	25534	15205	9831	9502	1524	286	2508	4381
gm	12802	9488	25680	25216	1874	715	63	65
exiv2	7016	5661	31201	31062	712	342	67	59
bison	5019	4419	6703	6700	387	234	57	65
Geo mean	3985	2466	5596	5548	458	134	145	156
% (w.r.t. CodeCov)	162%	100%	101%	100%	342%	100%	93%	100%

Table 3: Median number of testcases stored in the fuzzers' queues, edges covered and checks violated by such testcases, and average of the executions per second over 5 trials of 48h.

the original, un-instrumented program collectively exercised by executing the seeds (i.e., testcases) from the queues of InvsCov and CodeCov. Such coverage is a common metric in fuzzers evaluation, as a fuzzer cannot reveal a bug in a program point if it first does not explore it at least one time.

In Table 3 (column 'Edges') we report the median edge coverage of AFL++ when using, respectively, invariants or standard edge coverage as feedback. Overall, the differences are very small. For most targets, InvsCov results in a coverage comparable to CodeCov, showing that our technique does not result in a decrease of edge coverage. On some programs, our approach even helped the fuzzer to increase coverage over the saturated corpus, suggesting that some code paths may be reached only with the right combination of conditions over some program state variables.

It is important to remember that the goal of our system is NOT to increase code coverage, but instead to increase the state coverage along the paths reached by a fuzzer. Therefore, we study the number of invariants violated by using our feedback mechanism compared to the traditional CodeCov, as a proxy of the improved program state coverage. The 'Violated Checks' column in Table 3 shows that AFL++ with InvsCov, thanks to our instrumentation mechanism (§4.2), maintains a set of testcases that violate more invariants than AFL++ with just CodeCov. Overall, our approach was 3.4x more effective than pure CodeCov at helping the fuzzer to visit different partitions of the program state.

5.4 RQ4: Bug Detection

As the ultimate goal of Fuzz Testing is to detect bugs in programs we now analyze in more details the bugs InvsCov could find in our experiments and study their properties.

To compare InvsCov against classic edge coverage, we consider additional AFL++ configurations that exercise different designs in other components of the fuzzer, such as the scheduling strategies for mutations or seed selection. These

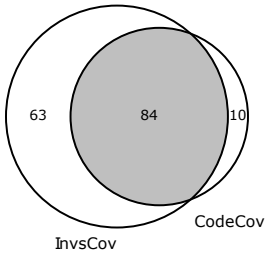


Figure 3: Venn Diagram showing the bugs Default found by either InvsCov or CodeCov, and by both (gray).

strategies are orthogonal to the feedback function in use. Therefore, in the end, we expect InvsCov to outperform CodeCov independently of other parameters. We believe this type of multi-pronged experiments allows for a more fair evaluation to isolate the contribution of the feedback technique alone.

In particular, we selected three AFL++ configurations for our tests:

- Default, i.e., the standard configuration of AFL++ used also for the other research questions;
- MOpt, i.e., AFL++ equipped with the MOpt [47] mutation scheduler, a powerful technique that dynamically prioritizes mutations according to their expected efficiency at any time in the execution;
- Rare, i.e., AFL++ scheduling that prioritizes seeds that exercise paths not along the 'hot' regions traversed by most seeds in the queue. Different, complex embodiments of this idea proved to be effective in a number of previous works [10] [44] [8].

We run these three state-of-the-art fuzzers on all target programs for 48h, to simulate a fuzzing campaign with a medium-small length, as previously used to evaluate fuzzers in works such as [58] and [6].

For crash de-duplication, we first grouped the reported crashes by using a standard call-stack hash from the stack

Program	DEFAULT			MOPT			RARE		
	INVSCOV	CODECOV	⋂	INVSCOV	CODECOV	⋂	INVSCOV	CODECOV	⋂
catppt	3	3	3	3	3	3	3	3	3
xls2csv	17	15	13	18	17	15	17	16	14
jasper	7	5	5	8	5	4	8	4	4
sndfile-info	11	10	10	10	10	10	11	10	10
pcre2	77	35	28	81	52	36	80	48	38
gm	19	14	13	18	14	13	20	14	13
exiv2	8	7	7	8	7	7	8	7	7
bison	5	5	5	5	5	5	5	5	5
Total	147	94	84	151	113	93	152	107	94
% (w.r.t. CODECOV)	156 %	100 %	89 %	134 %	100 %	82 %	142 %	100 %	88 %

Table 4: Median unique bugs found with and without invariant-based feedback over 5 trials of 48h for each target program and three different fuzzers (DEFAULT, MOPT and RARE).

Program	Reached	INVSCOV \ CODECOV
catppt	0	0
xls2csv	0	4
jasper	1	2
sndfile-info	1	1
pcre2	41	51
gm	0	6
exiv2	0	1
bison	0	0
Total	43	65

Table 5: Median number of bugs in the set difference between the INVSCOV and CODECOV bugs (see Table 4) that are reached in coverage by CODECOV but not triggered.

trace. However, as automatic de-duplication with stack hashes is generally unsound (it can both under- and over-count [42] depending on the case), we decided to manually inspect and triage each testcase.

Table 4 reports how many *unique manually deduplicated* bugs each fuzzer found over our set of subject programs[4]. The table also reports the intersection between the bugs found with our approach and with classic edge coverage. This relationship, summarized for all programs in the Venn diagram of Figure 3, highlights that guiding fuzzers by using state invariants not only results in more bugs being discovered, but also in *different* bugs[5].

Notably, the fuzzers that use our INVSCOV feedback never underperformed with respect to the corresponding CODECOV versions, in all configurations. For two targets, `catppt` and `bison`, all fuzzers found the very same number of bugs, suggesting that these bugs are easy to trigger without particular requirements over the program state. Notably, on some of the targets (`sndfile-info`, `xls2csv`, `gm`, `exiv2`), the use of invariants allowed the fuzzer to also discover previously unknown bugs and vulnerabilities, like the one we used as running example in §3.

To better understand the bugs that only INVSCOV was able to uncover, we classify them according to whether or not CODECOV was able to reach the crash point (obviously, without triggering it). We report the number of 'covered but not triggered' bugs for CODECOV in Table 5. It is interesting to observe that the instructions responsible for 43 of the 65 bugs discovered by INVSCOV were reached by CODECOV, but not triggered due to the lack of the correct combination of state conditions required to trigger the bug upon reaching the flawed program point. These types of bugs are particularly common for `pcre2`. Since the program is essentially a parser that makes use of lookup tables, its program states are heavily data-dependent. The importance of the program state is also confirmed by the fact that some of the crashing locations were reached already by the initial input corpus we supplied to the fuzzers. Thus, our approach shows a clear advantage for those programs that contain data dependencies in their flows.

For the remaining 21 bugs for which traditional fuzzers were unable to even reach the vulnerable location, a possible reason—as we discussed already for Table 3 (§5.3)—could be the fact that the use of invariants also allowed the fuzzer to achieve a slightly better code coverage. Indeed, specific conditions on program state values may be needed not only to uncover a fault but also to progress the exploration towards some code regions of the program under test.

As an additional set of experiments, we analyzed the default configuration of AFL++ with edge coverage augmented by context-sensitivity[6] (CTXCOV), firstly introduced by Chen

[4]The classes we observe are the typical ones from sanitization with ASAN and UBSAN (e.g., heap and stack overflow, division by zero). As the bugs are many, we omit tedious information on their types for brevity.

[5]INVSCOV may also miss bugs reported by CODECOV within a fixed time budget because of fuzzing entropy and different seed scheduling choices over different queues. However, those bugs are still within reach for INVSCOV.

[6]Fuzzers can use calling-context information, i.e., the sequence of routine calls concurrently active on the stack when reaching a program location [18].

et al. [11], which turned out to be the form of feedback that revealed more bugs in the recent analysis of Wang et al. [79]. We report the number of triaged bugs for InvsCov and CtxCov in Table 6, running five 48-h trials with the same initial corpus of the other experiments.

Our experiments confirm that CtxCov performs better than CodeCov (+11%), revealing more unique bugs on four targets (2 on `xls2csv`, 6 on `pcre2`, 1 on `exiv2` and `gm`). Nonetheless, call-stack information for the context does not contain explicit information on program data, and InvsCov consistently finds more or different bugs than CtxCov as well (e.g., +47 on `pcre2`, +6 on `gm2`). The number of bugs found by both slightly improves for two subjects (2 bugs on `xls2cov` and `pcre2`) compared to CodeCov, suggesting that calling-context information offered AFL++ a different angle based on call paths to exercise the program states that trigger such bugs.

Finally, we also explored the hybrid scenario (marked as *Combined* in Table 6) in which we augmented edge coverage with both our invariants and context-sensitivity at once. This combined approach led to the discovery of another heap vulnerability in `libsndfile` (function `wavlike_ima_decode_block`). While this solution performs overall slightly worse than invariants alone, we observed promising peaks on single runs of `pcre2` (119 for Combined, 92 for InvsCov, 47 for CtxCov), `jasper` (12-8-6), and `sndfile-info` (12-11-10). The downside of combining multiple feedback refinements is, in fact, that with larger queues (e.g., +79% on `pcre2` w.r.t. InvsCov) the randomness in seed scheduling impacts which program portions, and ultimately bugs, get explored in a limited time budget. We report the complete experimental data in Appendix §A, and leave the investigation of how to optimize combinations of this kind to future work.

5.5 RQ5: Run-Time Overhead

As our technique requires adding a more complex instrumentation to the program under test, it is reasonable to expect a higher run-time overhead with respect to CodeCov. Following the approach of the authors of RedQueen [5], we measured the average execution speed of AFL++ when executed on our target programs for 48h. Table 3 details (in column 'Exec / Sec') how many executions per second InvsCov and CodeCov were able to perform. The experiments show that our technique introduced on average a slowdown of 8%. We believe this to be a moderate price to pay to increase the ability of fuzzers to explore more (and more diverse) states of the programs under test.

A counter-intuitive result here is that for some programs the execution speed measured for InvsCov is higher than for CodeCov. The reason is that in some programs many invariant violations were triggered along fast code paths: as InvsCov causes the fuzzer to spend more time on the same code path if one or more invariants are violated along it, the fuzzer ultimately focused on those parts and executed the other, slower paths less often than when using CodeCov, thus benefiting from shorter executions.

Program	InvsCov	CtxCov	∩	Combined
`catppt`	3	3	3	4
`xls2csv`	17	17	15	18
`jasper`	7	5	5	6
`sndfile-info`	11	10	10	11
`pcre2`	77	41	30	65
`gm`	19	15	13	21
`exiv2`	8	8	7	8
`bison`	5	5	5	5
Total	147	104	88	138
% (w.r.t. CtxCov)	141 %	100 %	85 %	133%
% (w.r.t. CodeCov)	156 %	111 %	94 %	147%

Table 6: Median number of bugs found with InvsCov and CtxCov, their intersection, and the bugs found with a fuzzer Combined that uses both feedbacks simultaneously.

5.6 Discussion

The results of our experiments confirmed that our feedback, by distinguishing when program variables deviate from their 'usual' values, improves the sensitivity of a fuzzer for program states that code coverage alone fails to reward. Out of the 65 buggy program points that only our approach could drive to a crash, edge coverage alone was able to reach 43 of them, without however exposing the bug because the program was not in the correct state. Even when using refined code-based feedbacks like context-sensitivity, our approach continued to reveal more and different bugs than CGF.

Our tests also show that our instrumentation is tenable: it introduces only a moderate 62% growth on the fuzzer's queue size (orders of magnitude less than memory feedbacks, and still smaller than several code-based feedbacks [79]) and it slows down testcase execution by 8% on average. These costs are clearly amortized in our experiments by the many more unique bugs reported by our technique.

Finally, our feedback is not decremental in terms of code coverage compared to edge coverage and, in some cases, it can also 'unlock' more state-dependent program portions for further exploration. As briefly experimented in the *Combined* scenario, InvsCov and fine-grained forms of code feedbacks may also complement each other. Such a fuzzer would be able to better differentiate and explore those local state properties that are influenced by control-flow facts (e.g., the call path).

6 Other Related Works

This section covers security-related literature that makes use of invariants, and techniques orthogonal to our approach that improve the effectiveness of fuzzing explorations.

6.1 Invariants

Invariants historically play a key role in many development tasks such as software testing, optimization, and maintenance [26]. In the context of security research, several works have explored invariants for other problems as well.

In the context of anomaly detection, invariants can act as oracles for program hardening. Works such as [32] and [75] instrument programs to block memory corruption exploits in production, as run-time checking costs turn out to be modest. Web applications can benefit from similar protection as well, as explored in [15] with DAIKON and PHP code.

Fault localization is another popular twist. Whenever multiple invariants turn out to be violated, a typical workflow to locate the root cause is to study similar inputs to filter out non-relevant invariants. Some examples are [71], which uses dynamic backward slicing to remove more invariants, and [7], which employs a statistical analysis of the learned predicates.

Finally, in §2.1 we have mentioned how invariants in the form of specifications are pivotal for property-based testing. QUICKCHECK [12] is probably the most well-known among such systems. Recently works such as ZEST [59] and HYPOTHESIS [48] borrows fuzzing concepts like feedback-driven mutations to improve their efficiency when testing, respectively, Java and Python codebases.

6.2 Fuzz Testing

Fuzz Testing is an area of wide interest and intense investigation for academia and industry, with the number of works that try to improve individual aspects of its techniques growing massively every year [50]. We believe that, alongside the feedback mechanisms we discussed in §2.2, the most significant improvements have involved bypassing techniques for roadblocks to code coverage, mutation-based generation for exercising deeper program paths, and catching silent faults.

Luckily, these advancements are orthogonal to our approach. In §5.3 we already combined, off-the-shelf, several designs with our ideas, leveraging the flexibility of AFL++.

Roadblocks. Roadblocks are comparison sequences over the input that standard mutations can hardly satisfy, hence they limit code exploration. Magic values and checksum fields in input formats are the two main kinds of roadblocks.

As magic values typically undergo multi-byte comparisons, and classic structure-blind mutators treat the input as a stream of bytes, matching all the involved bytes is extremely unlikely. Solutions may come from using a special-purpose feedback for partial progress at comparisons [2] [45], concolic execution for white-box fuzzing [82] [66], or techniques that extract comparison operands to replace input portions [69] [5].

Typically used for validation in binary formats, checksums are even more difficult to overcome. Known solutions involve format-specific mutators or code transformations that detect and temporarily override checksum checks [5] [63] [28].

Valid Inputs. General-purpose fuzzers can see high rates of invalid generated inputs. Such inputs typically end up exercising code from early parsing stages of a program, failing to reach deeper regions. For exploring such regions effectively, a fuzzer must thus focus on producing valid inputs.

Works like [65] [3] take a hand-written input format specification to guide their mutator. Later works tried to automatically learn an approximate one [6] [28] [53] [81].

A different approach is to constrain the mutator to preserve input portions that conduct to deeper paths and to further restrict the values that other input fields can take along them using, for instance, concolic tracing information [41].

Catching Silent Faults. Most fuzzers seek to expose faults that may be potential vulnerabilities, yet oftentimes a fault does not trigger a directly observable failure. For instance, a one-byte heap read overflow is unlikely to crash a C program.

To catch such bugs, fuzzing users instrument the program under test with additional tripwires to expose silent faults. For instance, source-based fuzzers offer the possibility to instrument programs with sanitizers like ASAN [74]. Others make use of binary-only tripwires to uncover silent corruptions [55], inserted dynamically [29] or statically [21].

Nonetheless, current sanitizers cannot catch some pure-logic bugs. In the lack of hand-written assertions, fuzzing to uncover such bugs automatically is an open field of research.

Hard Targets. Instrumentation, execution, and testcase administration are far-from-trivial problems when working with hard targets [72]. As our approach is concerned only with the first, our technique is general enough to work with binary-only frameworks (e.g., binary rewriters, dynamic translators) that can expose program state values. Binary-level is the only option when dealing with closed-source targets, like firmware images executing under whole-system emulation with emulated peripherals [84] [64] or with re-hosting [13] [49]. More invariant pruning heuristics may, however, be needed in the absence of debug symbols. Our approach would not be compatible, instead, with simpler schemes that are breakpoint-based [56] or use hw-assisted tracing for control flows [73].

7 Limitations and Future Directions

Our approach augments the classic edge coverage feedback with information on violated likely invariants. As we emit AFL-compliant instrumentation for the sake of compatibility, there is a possibility of hash collisions—just like with AFL—when indexing the shared map for coverage updates. AFL++ offers an alternate link-time instrumentation scheme [39] that is collision-free but breaks compatibility. We may devise an INVSCOV variant that benefits from such design, and possibly explores also split maps for the invariant and edge feedbacks.

On the methodological side, an intrinsic limitation of our approach is that it is not adaptive. We learn invariants once, while there could be potential to explore by refining them

as the exploration advances and new value conditions are observed. A follow-up of our approach would then be to devise an online invariant mining module. Recent machine learning advances in anomaly detection like [16] could offer valid support to this end. We believe that a fuzzer that adaptively learns the state space partitions over variables can have a positive practical impact, and potentially help in desaturating fuzzing campaigns like OSS-Fuzz to catch more bugs in software already well-tested with CGF solutions.

Finally, as we study data facts at basic-block level, our likely invariants cannot capture 'implicit' relations between variables that do not get processed together in any block.

8 Conclusion

Using likely invariants as feedback for fuzzers brings novel ideas to better abstract, and in turn explore program states. We argued that some bugs may be readily discovered by taking into account program state conditions that control flow alone does not entail, accessing seldom-explored corner cases where vulnerabilities may lie undetected for a long time. We achieved this goal without incurring the state explosion problem and with a moderate performance overhead, amortized by an increased number of found bugs. We hope that our work can pave the way for more research on program state approximations to serve as feedback for Fuzz Testing.

Acknowledgments

We would like to thank our shepherd Sooel Son and the anonymous reviewers for their constructive feedback. We would also like to thank Slasti Mormanti for his valuable insights. This project has been supported by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA875019C0003.

References

[1] Kvasir C/C++ front end. https://plse.cs.washington.edu/daikon/download/doc/daikon.html#Kvasir. [Online; accessed 10 Jan. 2021].

[2] Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016. [Online; accessed 10 Jan. 2021].

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[4] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. IJON: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: fuzzing with input-to-state correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[6] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1985–2002. USENIX Association, August 2019.

[7] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. AURORA: Statistical crash analysis for automated root cause explanation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 235–252. USENIX Association, August 2020.

[8] Marcel Böhme, Valentin Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 14th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 1–11, 2020.

[9] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2329–2344. Association for Computing Machinery, 2017.

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043. Association for Computing Machinery, 2016.

[11] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[12] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN Int. Conference on Functional Programming*, ICFP '00, pages 268–279. Association for Computing Machinery, 2000.

[13] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias

Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1201–1218. USENIX Association, August 2020.

[14] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[15] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 63–86, September 5–7, 2007.

[16] Thomas Defard, Aleksandr Setkov, Angelique Loesch, and Romaric Audigier. PaDiM: A patch distribution modeling framework for anomaly detection and localization. In *Pattern Recognition. ICPR International Workshops and Challenges*, pages 475–489. Springer International Publishing, 2021.

[17] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, page 15–27. Association for Computing Machinery, 2019.

[18] Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. Mining hot calling contexts in small space. *Software: Practice and Experience*, 46(8):1131–1152, 2016.

[19] Jared D. DeMott and R. Enbody. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. Black Hat USA, 2007.

[20] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev.Ng: A unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 131–141. Association for Computing Machinery, 2017.

[21] S. Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[22] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. LAVA: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, 2016.

[23] M. Eddington. Peach fuzzing platform. https://web.archive.org/web/20180621074520/http://community.peachfuzzer.com/WhatIsPeach.html. [Online; accessed 10 Jan. 2021].

[24] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[25] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[26] Michael Dean Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, 2000. AAI9983472.

[27] G. Fink and K. Levitt. Property-based testing of privileged programs. In *Tenth Annual Computer Security Applications Conference*, pages 154–163, 1994.

[28] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020. Association for Computing Machinery, 2020.

[29] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing binaries for memory safety errors with QASan. In *2020 IEEE Secure Development Conference (SecDev)*, 2020.

[30] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[31] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2577–2594. USENIX Association, August 2020.

[32] C. Giuffrida, L. Cavallaro, and A.S. Tanenbaum. Practical automated vulnerability monitoring using program state invariants. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12. IEEE CS, 2013.

[33] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'08, 2008.

[34] Alex Groce and John Regehr. The Saturation Effect in Fuzzing. https://blog.regehr.org/archives/1796. [Online; accessed 10 Jan. 2021].

[35] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301. Association for Computing Machinery, 2002.

[36] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, 3(03):243–250, may 1977.

[37] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[38] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), November 2020.

[39] Marc Heuse. afl-clang-lto - collision free instrumentation at link time. https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.lto.md, 2020. [Online; accessed 10 Jan. 2021].

[40] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458. USENIX Association, August 2012.

[41] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1613–1627. IEEE Computer Society, 2020.

[42] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2123–2138. Association for Computing Machinery, 2018.

[43] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the 2004 Int. Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[44] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 475–485. Association for Computing Machinery, 2018.

[45] LLVM Project. LibFuzzer - Value Profile. https://llvm.org/docs/LibFuzzer.html#value-profile. [Online; accessed 10 Jan. 2021].

[46] LLVM Project. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, September 2018. [Online; accessed 10 Jan. 2021].

[47] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966. USENIX Association, August 2019.

[48] David R. MacIver, Zac Hatfield-Dodds, and Many Other Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.

[49] Dominik Maier, Benedikt Radtke, and Bastian Harren. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. USENIX Association, August 2019.

[50] V. Manes, H. Han, C. Han, S.K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, (01), oct 5555.

[51] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 1024–1036. Association for Computing Machinery, 2020.

[52] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. *Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing*, page 701–712. Association for Computing Machinery, 2020.

[53] Björn Mathis, Rahul Gopinath, and Andreas Zeller. Learning input tokens for effective fuzzing. In *ISSTA '20: 29th ACM SIGSOFT Int. Symposium on Software Testing and Analysis*, pages 27–37. ACM, 2020.

[54] B. Miller, M. Zhang, and E. Heymann. The relevance of classic fuzz testing: Have we solved this one? *IEEE Transactions on Software Engineering*, 2020.

[55] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium*, 2018.

[56] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[57] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 47–62. USENIX Association, October 2020.

[58] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security*, August 2020.

[59] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proc. of the 28th ACM SIGSOFT Int. Symposium on Software Testing and Analysis*, ISSTA 2019, pages 329–340. Association for Computing Machinery, 2019.

[60] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. FuzzFactory: Domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[61] Karthik Pattabiraman, Giancinto Paolo Saggese, Daniel Chen, Zbigniew Kalbarczyk, and Ravishankar Iyer. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Transactions on Dependable and Secure Computing*, 8(5):640–655, 2010.

[62] Mathias Payer. The fuzzing hype-train: How random testing triggers thousands of crashes. *IEEE Security and Privacy*, 17(1):78–82, 2019.

[63] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710, May 2018.

[64] Hui Peng and Mathias Payer. Usbfuzz: A framework for fuzzing USB drivers by device emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2559–2575. USENIX Association, August 2020.

[65] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[66] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198. USENIX Association, August 2020.

[67] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 133–138. Association for Computing Machinery, 1959.

[68] Fernando Magno Quintao Pereira, Raphael Ernani Rodrigues, and Victor Hugo Sperle Campos. A fast and low-overhead technique to secure programs against integer overflows. In *Proc. of the 2013 IEEE/ACM Int. Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11. IEEE Computer Society, 2013.

[69] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[70] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27. Association for Computing Machinery, 1988.

[71] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *Proc. of the 18th Int. Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 139–152. Association for Computing Machinery, 2013.

[72] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[73] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *Proc. of the 26th USENIX Security Symposium*, SEC'17, pages 167–182. USENIX Association, 2017.

[74] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, page 28. USENIX Association, 2012.

[75] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. *NDSS*, 2016.

[76] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295, 2019.

[77] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[78] Guido Vranken. Cryptofuzz - differential cryptography fuzzing. https://github.com/guidovranken/cryptofuzz, 2019. [Online; accessed 10 Jan. 2021].

[79] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15. USENIX Association, September 2019.

[80] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294. Association for Computing Machinery, 2011.

[81] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang. SLF: Fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 712–723, 2019.

[82] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 745–761. USENIX Association, 2018.

[83] Michał Zalewski. American Fuzzy Lop - Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016. [Online; accessed 10 Jan. 2021].

[84] 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 *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114. USENIX Association, August 2019.

A Appendix

In the following we report supplementary data for some of the experiments that we discussed throughout §5.

For the first research question, alongside the total number of invariants we also collected frequency metrics for probes at function and basic-block level. Due to the dynamic nature of the invariant mining process, only code actually reached in any execution from the input corpus can feature invariant checks. Table 7 reports figures computed over reached code only and with all our pruning optimizations enabled. While the code characteristics (most prominently, the varying complexity of individual functions) are reflected by heterogeneous values for invariants checked by a single function, when considering basic blocks we observe rather regular trends, with two peaks for `bison` and `jasper`, due to their basic blocks typically longer and richer of LLVM IR virtual register manipulations involving variables of interest for our method.

For the last set of experiments that we conducted for studying our bug finding capabilities, here we report statistics on the median queue size for the CtxCov and Combined fuzzers (Table 8), and the peak number of unique bugs identified among the 5 runs we made for each program under test (Table 9). The context-sensitive feedback benefited from our invariants as well, yet a larger queue impacts the program states explored by different runs within the 48h budget.

Program	Per Function	Per Block	Total
`catppt`	9.78 (14)	1 (137)	137
`xls2csv`	12.12 (33)	1.15 (349)	400
`jasper`	29.88 (306)	2.94 (3106)	9144
`sndfile-info`	20.09 (150)	1.01 (2979)	3013
`pcre2`	106.73 (45)	1.32 (3651)	4803
`gm`	28.78 (499)	1.38 (10391)	14362
`exiv2`	5.31 (1042)	1.17 (4708)	5534
`bison`	27.23 (230)	2.14 (2922)	6263
Geo mean	20.53	1.41	2784

Table 7: Average number of produced invariants for each function and basic block with at least one invariant. The reference baseline is reported between parentheses.

Program	InvsCov	CtxCov	Combined
`catppt`	213	149	281
`xls2csv`	1358	950	1766
`jasper`	10831	3528	18057
`sndfile-info`	1764	1525	2096
`pcre2`	25534	27227	45705
`gm`	12802	10928	14302
`exiv2`	7016	8457	9073
`bison`	5019	4975	6076
Geo mean	3985	3143	5356

Table 8: Median number of testcases in the queues of InvsCov, CtxCov, and Combined.

Program	InvsCov	CtxCov	Combined
`catppt`	4	3	4
`xls2csv`	19	19	19
`jasper`	8	6	12
`sndfile-info`	11	10	12
`pcre2`	92	47	119
`gm`	22	17	21
`exiv2`	8	8	8
`bison`	6	5	6
Total	170	115	201

Table 9: Peak number of unique bugs found by InvsCov, CtxCov, and Combined among the 5 runs.