Data Coverage for Guided Fuzzing

Mingzhe Wang[1], Jie Liang[1], Chijin Zhou[1], Zhiyong Wu[1], Jingzhou Fu[1], Zhuo Su[1],
Qing Liao[2], Bin Gu[3], Bodong Wu[4], and Yu Jiang[1]*
[1]*Tsinghua University,* [2]*Harbin Institute of Technology,*
[3]*Beijing Institute of Control Engineering,* [4]*Huawei Technologies Co., Ltd*

Abstract

Code coverage is crucial for fuzzing. It helps fuzzers identify areas of a program that have not been explored, which are often the most likely to contain bugs. However, code coverage only reflects a small part of a program's structure. Many crucial program constructs, such as constraints, automata, and Turing-complete domain-specific languages, are embedded in a program as constant data. Since this data cannot be effectively reflected by code coverage, it remains a major challenge for modern fuzzing practices.

To address this challenge, we propose data coverage for guided fuzzing. The idea is to detect novel constant data references and maximize their coverage. However, the widespread use of constant data can significantly impact fuzzing throughput if not handled carefully. To overcome this issue, we optimize for real-world fuzzing practices by classifying data access according to semantics and designing customized collection strategies. We also develop novel storage and utilization techniques for improved fuzzing efficiency. Finally, we enhance libFuzzer with data coverage and submit it to Google's FuzzBench for evaluation. Our approach outperforms many state-of-the-art fuzzers and achieves the best coverage score in the experiment. Furthermore, we have discovered 28 previously-unknown bugs on OSS-Fuzz projects that were well-fuzzed using code coverage.

1 Introduction

Code coverage is an essential part to software testing. In any case, code must be executed to trigger the underlying bug. This is particularly important in the case of guided fuzzing, a type of automated testing that identifies potential bugs and vulnerabilities by maximizing code coverage with novel test cases. The use of code coverage in guided fuzzing has been shown to be highly effective, leading to its widespread adoption in both industry [30,35] and academia [29,34,38]. In fact, the use of code coverage in guided fuzzing has been responsible for the discovery of over 40,000 bugs in 650 popular projects [1]. Most academic papers on fuzzing also rely on code coverage as a key component of their testing strategy.

However, code coverage is *not* sufficient to fully reflect a program's semantics. As Niklaus Wirth [36] pointed out, "algorithms + data structures = programs." In other words, *programs* are concrete formulations of abstract *algorithms* based on particular representations and structures of *data* [36]. This can also be seen from the perspective of programming language implementation: when a compiler converts source code into machine code, the algorithms are transformed into *machine instructions*, while the data structures are encoded as *constant data*.

In addition to code, constant data plays a crucial role in program constructs. There are two physical forms of constant data: immediate values and static values.

- *Immediate values* are simple, directly embedded in code, and commonly found in programs. For example, in the C programming language, the code `*ptr = 0x1234abcd` would lower to the x86 machine code c7 00 cd ab 34 12, with the immediate value 0x1234abcd embedded as the last four bytes of the machine code.

- *Static values* are more complex and expressive. In the C programming language, string literals, `static` variables, and global variables are all examples of static values. These values can encode complex structures and convey intricate semantics, including arrays, linked lists, lookup tables, and even graphs.

Data-intensive program constructs are often difficult to test with code coverage guided fuzzing. To illustrate this point, consider the scenario of testing a parser for a highly-structured input format such as SQL. In this case, the programmer writes a grammar specification and uses a tool like YACC to automatically generate the parser code. The transition rules derived from the grammar are encoded as large arrays, while a small

*Corresponding author.

piece of template code is used to drive the transition. Therefore, code coverage only reflects the superficial logic of "how to run *any* automata," while data coverage reflects the real logic of "how *this* SQL automata runs." This is why data coverage is so important for fuzzing: it allows fuzzers to fully test data-intensive program constructs.

According to the definition of constant data symbols, data coverage can have varying granularities, such as variable, array, field, or bit coverage, similar to code coverage's function, line, and branch coverage. However, a trade-off exists between the precision of the data coverage measurement and the overhead it introduces. For instance, achieving perfect data coverage would require symbolizing the program's address space into a bit vector and collecting the constant data symbols that contribute to the program's output. While this would provide highly precise measurements, it also significantly slow down the program's execution and reduce the overall fuzzing throughput.

We optimize data coverage for real-world fuzzing practices. Our techniques aim to retain accuracy while maintaining high performance. They consist of three stages:

1. Before fuzzing begins, we perform static analysis and instrument potential data accesses. To remove unnecessary instrumentation, we divide data accesses into 6 categories and instrument them differently.
2. During fuzzing, when a data access is intercepted, the runtime library translates the different kinds of data accesses into tuples of (address, length). We store the novel ones in a novel set for further inspection.
3. After the execution of a test case, the fuzzer checks the novel data coverage set. Code and data features are used to jointly adjust the fuzzer's exploration direction.

To demonstrate the effectiveness of data coverage, we implemented a reference fuzzer based on libFuzzer and conducted third-party evaluation on Google's FuzzBench. Our evaluation incorporates many classic fuzzers and fully replicates the original FuzzBench paper's setup [23]. The results indicate that data coverage significantly boosts libFuzzer's normalized coverage score from 87.65 to 98.31, resulting in an improved rank for the fuzzer from 9th place to 1st place among the 12 fuzzers tested. Interestingly, we found that the introduction of data coverage alone outperforms the combination of many advanced fuzzing strategies, including AFL++ [9], the default fuzzer for Google's continuous fuzzing service OSS-Fuzz.

Moreover, when we applied data coverage to real-world programs on OSS-Fuzz, we discovered 28 previously-unknown bugs despite continuous fuzzing by thousands of machines in Google's cluster. These findings indicate that data coverage can uncover novel program states and bugs even when a program is continuously fuzzed with code coverage guidance.

In summary, this paper makes the following contributions:

- We propose a data coverage approach for guided fuzzing. It helps to solve the limitations of code coverage when applied to data-intensive program constructs.
- We present optimization techniques for using data coverage in fuzzing, including novel methods for coverage maintenance, representation, and utilization.
- We implement a reference fuzzer based on data coverage and evaluate its effectiveness. Experiments show that it achieves the highest coverage and stability among all the evaluated fuzzers. It also found 28 previously-unknown bugs on well-fuzzed OSS-Fuzz projects.

For open, reproducible research, we publish the source code[1], the raw experiment data[2], and the detailed evaluation report on FuzzBench[3].

2 Towards Data Coverage

2.1 Limitations of Code Coverage

Although code coverage is a valuable tool for fuzzing, it has limitations when it comes to exploring data-intensive program constructs. One such construct involves immediate values used in predicates, which determine the truth value of a proposition. A common example of this type of construct can be found in the widely-used color management engine, LCMS, which accepts ICC color profiles as input data. As illustrated in Figure 1, LCMS performs a sanity check on the ICC file by verifying the magic number. While the predicate `magic != 0x61637370` is relatively simple to understand and reason about, using code coverage to explore it can be challenging.

```
#define cmsMagicNumber  0x61637370     // 'acsp'

// Validate file as an ICC profile
if (_cmsAdjustEndianess32(Header.magic) != cmsMagicNumber) {
  cmsSignalError(... "not an ICC profile, invalid signature");
  return FALSE;
}
```

Figure 1: Predicate in LCMS testing for magic numbers.

In the code snippet provided, code coverage can only determine whether the check passes or fails. However, when the check fails, the code coverage remains the same regardless of how close the guessed value is to the correct value. Without effective guidance, the fuzzer must "guess" the values blindly, which means that the likelihood of a successful guess is 2^{-32}. In other words, to achieve a success rate of 50%, the fuzzer

[1]The source code is available at https://github.com/THU-WingTecher/wingfuzz.

[2]The experiment data is available at https://console.cloud.google.com/storage/browser/fuzzbench-data/2022-10-08-wingfuzz.

[3]The evaluation report is available at https://www.fuzzbench.com/reports/experimental/2022-10-08-wingfuzz.

would need to try 3×10^9 times, which would take over a month assuming 1,000 executions per second.

2.2 Limitations of Constraint Solving

Constraint solving is an effective way to solve simple branches. Two popular solving approaches are concolic execution [7, 27, 33] and intelligent branch solving [3, 19, 20, 28].

Concolic execution represents the constraint as symbolic expressions, enabling solvers such as SMT solvers to solve them. We can represent the input bytes as a sequence of 8-bit vectors, with each byte declared as SMT-LIB clause `(declare-fun inputX () (_ BitVec 8))`. As the program executes, we gradually update the symbolic representations of all values based on the `inputX` variables. To solve the failed predicate, we add an assertion that mandates all bytes to be equal to the expected values: `(assert (and (= input0 #x61) ...))`. By querying this assertion, we request the SMT solver to provide concrete input values satisfying the constraints.

Intelligent branch solving identifies which input bytes are used by the constraint and employs a tailored mutation strategy on these bytes [3, 26]. For instance, to solve the branch in Figure 1, we can record the operands of the failed predicates and match them against the input data. In this example, we find that the first 4 bytes of the input impact the branch and are expected to be `0x61637370`. Therefore, we can simply replace the bytes with the expected value to solve the branch.

However, there are limitations for data-intensive program constructs, particularly for static values with rich semantics. For static values, it is inefficient for solvers to resolve one constraint. Moreover, the fuzzer may fail when the semantics of static values are not constraints.

2.2.1 Inefficiency in Constraint Solving

Even simple string comparisons can be challenging for constraint solvers. For example, LCMS implements case-insensitive string comparison on its own for portability, as shown in Figure 1. Neither concolic execution nor intelligent branch solving can effectively solve this constraint.

```
int cmsstrcasecmp(const char* s1, const char* s2) {
  // [...]
  while (toupper(*us1) == toupper(*us2++))
    if (*us1++ == '\0')
      return 0;
}
```

Figure 2: Manual string comparison in LCMS.

Concolic execution can solve this constraint by enumerating paths in the program systematically, but it fails to distinguish useful differences among paths from superficial ones. For instance, in this case, the `toupper` function may contain several branches to handle the non-ASCII, lower-cased, or upper-cased scenarios. Thus, there could be 8 paths to handle in one iteration, assuming two cases for the first `toupper`, two cases for the next `toupper`, and two cases for the equality comparison. If the loop runs for m iterations, there would be 8^m paths to solve! Even with modern optimizations that merge similar paths, it can still incur high overhead in symbolization and solver invocations.

Table 1: Hard-Coded Buffer Comparison Routines in AFL++

Program	Count	Example
C++	4	`operator==(string&, char*)`
C	10	`strcmp`
apache	3	`ap_cstr_casecmp`
busybox	1	`memcmpct`
curl	5	`Curl_safe_strcasecompare`
glib	9	`g_strcasecmp`
lcms	1	`cmsstrcasecmp`
libxml	7	`xmlStrcasecmp`
openssl	4	`OPENSSL_memcmp`
samba	1	`strcsequal`
Total	45	

Intelligent branch solving cannot solve this constraint without human intervention. Because the comparison of a single character is easy to succeed, the `while` branch is quickly marked as solved and not inspected further. Consequently, the constraint for the remaining characters is never passed to the solver. As a remedy, state-of-the-art fuzzers tend to work around this issue with per-program optimizations. For instance, AFL++ implements "CmpLog" instrumentation to record the operands of failed comparison routines to solve them with heuristics. Rather than instrumenting individual branches in a program-agnostic manner, it hard-codes a curated list of comparison routines and treats each invocation as an abstract branch. As shown in Table 1, more than two-thirds of the optimizations are focused on individual programs. Obviously, this approach does not scale.

In conclusion, constraint solvers cannot effectively solve static value-based branches. Although it is technically possible to solve this branch with concolic execution, it is inefficient in fuzzing practices. Intelligent branch solving also requires manual optimizations for individual programs.

2.2.2 Challenges in Non-Constraints

Complex structures, such as lookup tables, binary trees, and directed graphs, can be represented as static data. However, their semantics are not simple constraints, which makes them unsuitable for constraint solvers [2]. For example, libpcap supports customized packet filtering with user-provided expressions, which is implemented using a flex-based lexer and a bison-based parser. As Figure 3 shows, the lexer encodes the complex transition table in constant arrays and drives the machine using simple logic [18].

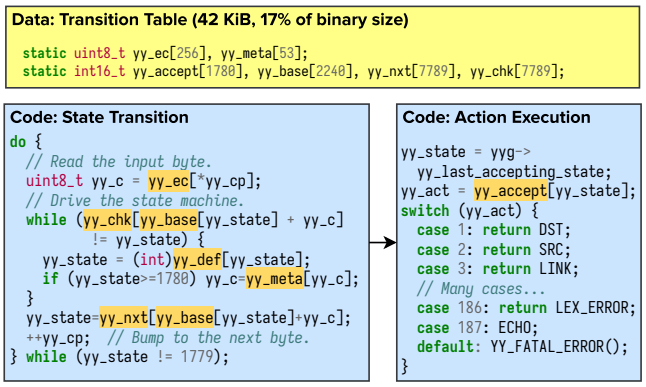

Figure 3: Finite-state machine implementation of libpcap. Based on the predefined *transition table* (shown in top), the *transition logic* (shown in the left) alters the machine's state according to input. When a final state is entered, the *execution logic* (shown in the right) performs user-defined actions. Data references in code are highlighted in yellow.

To explore the libpcap program, the fuzzer needs to trigger novel state transfers within the automaton, reach new accepted states, and execute new program logic in the end. However, the simplicity of the machine's logic results in saturated code coverage on the left and coverage plateau on the right. Since the fuzzer's coverage is stranded in the switch statement on the right, the fuzzer seeks help from the constraint solver. Concolic execution fails to symbolize indirect memory accesses, falling back to a concrete version of `yy_act`, which is unsolvable. Intelligent branch solving also fails since the predefined mutation strategy is inapplicable, as the value of `yy_act` is not directly related to input data. Therefore, none of the solvers can efficiently explore the state machine and trigger novel actions.

2.3 Benefits of Data Coverage

2.3.1 Benefits for Static Values

Static values are objects with static storage duration and predefined values. In the C programming language, string literals, `static` variables, and global variables are all examples of static values. A common implementation of these values is to convert the data to a binary form and store it in data segments. At runtime, these segments can be efficiently mapped to the process's address space.

The complexity of static values provides rich semantics. Complex structures, such as lookup tables, binary trees, and directed graphs, can be represented as static data. As demonstrated in the automaton example of Figure 3, the semantics of static values can be much more complex than constraints, where existing techniques such as constraint solving simply do not work. In this case, data coverage can be a crucial complement to code coverage. By tracking constant data usages, the fuzzer can distinguish 1,780 intermediate states and gradually explore them, executing novel actions as it does so.

2.3.2 Benefits for Immediate Values

Immediate values are simple values that are embedded directly inside instructions. For example, in the statement `*ptr = 0x1234abcd`, the value `0x1234abcd` is an immediate value. When this statement is compiled to x86 machine code, the instruction will be represented as c7 00 cd ab 34 12 (or `movl $0x1234abcd, (%eax)` in assembly). The immediate value is encoded as the last four bytes of the movl instruction. Immediate values can take up a significant amount of space in machine code. For example, in the above `movl` instruction, 67% of the machine representation is pure data.

Similar to constraint solving techniques, data coverage can be a crucial complement to code coverage. We can understand how data coverage helps to solve this scenario by the "short-circuiting" nature of predicates. Essentially, a predicate divides the input domain into two equivalence classes. Since the returned value is only one bit, not all bits of information are required to compute the result. In Figure 1's example, the least significant bit of immediate value `0x61637370` is 0. When the processor performs the comparison, it can immediately terminate the comparison and return "true" for any odd value (i.e., any value with a least significant bit of 1) because of the mismatch. This means that only one bit of the 32-bit immediate value is effectively used in the comparison.

To detect these subtle changes in integer predicates, we can use data coverage at bit-level precision. Specifically, we can guide fuzzing by maximizing the effectively used bits in predicates. When all 32 bits of the immediate value are effectively used, the predicate will flip and new code will emerge automatically.

3 Data Coverage in a Nutshell

Figure 4 illustrates how we guide fuzzing with data coverage. Before fuzzing starts, we instrument the program for various kinds of data access primitives. During fuzzing:

1. The fuzz engine generates a test case and executes the target program with it.
2. During the test case execution, the instrumented fuzz target records data coverage. In this case, the instrumentation code intercepts a 16-bit switch at 0x404a1c and passes its condition and case values to the runtime library.
3. The runtime abstracts the switch according to its semantics, treating it as two integer accesses. The first one (omitted in the figure) reads 4 bits from 0x404a19, while the next one (shown in the figure) reads 15 bits from 0x404a20.
4. With the access tuple, we detect novel coverage by comparing it to the known coverage database. In this case, the

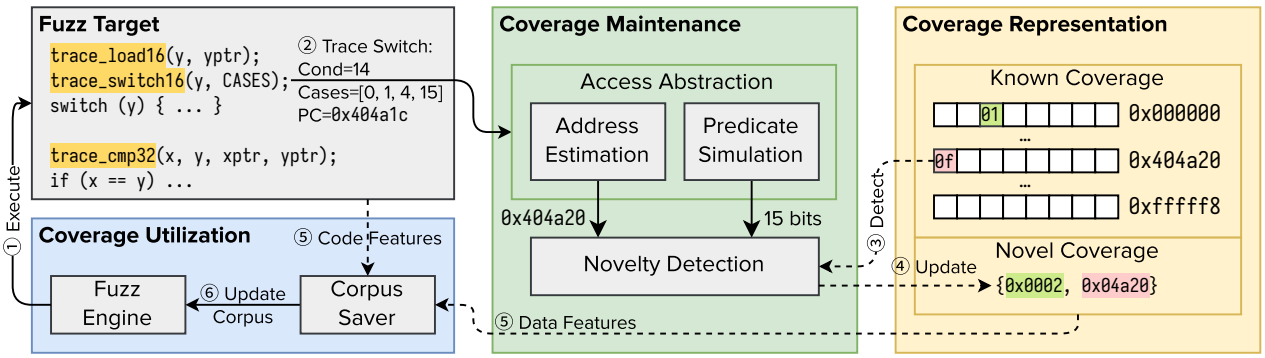

Figure 4: Fuzzing with data coverage. The *fuzz target* is instrumented during compilation. During program execution, the *coverage maintenance* component intercepts the program's data accesses and updates the coverage representation. After the test case execution completes, the *coverage utilization* component adjusts the fuzzing directions based on the novel coverage.

access to 0x404a20 improves the previously known coverage length to 15 bits. We add its address to the set of novel coverage and continue the program's normal execution.

5. After the test case execution is completed, we collect the complete code coverage and the set of novel data features.
6. The corpus saver analyzes both code and data features. In this case only novel data coverage is detected, thus the saver replaces an existing seed with the current test case.

3.1 Coverage Maintenance

We model each data access as a region, represented as a tuple of **(address, length)**. The address is based on byte granularity, whereas the length is measured in bits. We intercept and convert all the program's data accesses to such tuples. There are several special cases worth mentioning.

3.1.1 Load Filtering

Static values must be used by loading their values to registers. Therefore, we instrument load instructions to intercept static value usages. However, not all load operations are based on static values: the address can also be derived from the stack or the heap.

To filter load addresses, we analyze the runtime memory layout. The layout is obtained from the operating system's dynamic linker, which is responsible for loading a binary's segments to the program's address space and managing shared objects. For example, the dynamic linker of GNU libc provides the `dl_iterate_phdr` function to iterate through all the program headers (`dl_phdr_info`) of loaded objects. With this information, we can construct the runtime memory layout of the current process at the segment level, including the segment's type, address, and length. We use the layout to determine whether an address belongs to the program's binary. To improve efficiency, we pre-compute the valid memory range when the program starts. This design only requires two integer comparisons for each check.

3.1.2 Program Counter for Address Estimation

Immediate values are designed to be embedded in instructions. Typically, their addresses are implementation-defined and cannot be directly accessed. For example, the ARM64 architecture has a fixed 4-byte instruction encoding, which means that a 32-bit integer is spread across two instructions. Tracking the precise address of immediate values is costly and does not provide any benefits for fuzzing.

To estimate the address of immediate values, we use the *program counter*. Specifically, when instrumentation is needed, we insert a call to the runtime library *before* the parent instruction of the immediate value. When the program calls the runtime library, it saves the program counter in the manner specified by the application binary interface (ABI). Because the saved program counter is adjacent to the parent instruction and the contained immediate value, we read the value following the platform ABI and use it as an approximation of the immediate value's address.

3.1.3 Predicate Emulation

Over approximation may occur if we directly treat the data's storage bits as the access length. As discussed in §2.3.2, predicates compress the input domain into a boolean value. Therefore, not every bit of information is used in the decision process. For example, when testing `x == 0`, the processor may terminate the comparison as soon as a non-zero bit is found.

To determine the number of effectively used bits of information in predicates, we emulate predicates based on the following observation: predicates require more information to complete as the operands become more similar to each other. For example, when testing `0b0001 == 0b1110`, we can easily return "false" if *any* of the four bits is selected; however, when testing `0b1110 == 0b1110`, *every* bit must be compared to draw a conclusion. Therefore, *we define the "effectively-used bit" of equivalence relations as the number of equal bits in the corresponding positions for both operands.*

However, this measurement cannot be applied to partial order relations. Unlike equivalence relations, which treat all bits equally, the predicates of partial order relations scan from the most significant bit to the least significant bit. For example, when testing `0b0001 > 0b1110`, the result can be obtained by looking at the most significant bits alone. As a solution, we add a constraint to the original measurement, requiring that the measurement starts from the most significant bit. Therefore, *we define the "effectively-used bit" of partial order relations as the number of consecutive equal bits of both operands starting from the most significant bit.*

Since the measurement of effectively used bits is used frequently, we accelerate the operation with bit manipulation. Suppose that we are measuring the actually used bits between `x` and `y`, we first compute the different bits `diff = x xor y`. A bit in `diff` would be 0 if the corresponding bits in `x` and `y` are equal. Therefore, the final measurement would be the number of zeros in `diff` for equivalence relations and the number of *leading* zeros of `diff` partial order relations. We compute the results with the compiler intrinsic `__builtin_popcount` and `__builtin_clz` for potential hardware acceleration.

3.2 Coverage Representation

We represent the known coverage as an 8-bit integer array. For each access tuple, the lowest 24 bits of the address are used as the array index, and the length is stored directly as the element's value. For example, as shown in Figure 4, if the program reads 15 bits at 0x404a20, then the array's 0x404a20-th element is accessed to obtain the known access length. If the known length is less than 15, we update the value to 15 and mark the access as novel.

The design of known coverage reduces the overhead of instrumentation. First, most memory access primitives (e.g., load instructions and `memcpy`) touch multiple adjacent bytes. This design merges all accesses into one byte, reducing the extra memory accesses issued by instrumentation. Second, unnecessary predicate emulations are reduced.

Predicate emulation improves the tracking precision to the bit level, but slow bit manipulations are involved. We use `0xff` as a saturation mark to represent whether the maximum possible length has been reached. In this way, we can directly skip predicate simulation if the saturation mark is found.

We represent the novel coverage as a set, which contains the base addresses of novel access tuples. To minimize the impact on program execution, we implement the set as a dynamic-sized array. As the instrumentation logic identifies a novel access tuple during execution, the address is immediately appended to the array. Following each execution, we convert the array to a set through sorting and de-duplication processes.

The design of the novel coverage set removes the expensive post-execution coverage scan. Existing code coverage pipelines typically scan the coverage array after each execution to detect new coverage. If we reuse this pipeline to handle data coverage, then it would take too many cycles: to prevent collisions and the accompanying inaccuracy, we store the known coverage in a large 16 MiB array. Our new design solves this problem by delegating the maintenance of novel coverage to the instrumentation logic during program execution. Therefore, the post-execution coverage discovery is essentially an $O(1)$ access to the array's length.

3.3 Coverage Utilization

When a test case finishes execution, we determine the fuzzing exploration direction with both code and data features. Algorithm 1 presents the potential actions. The most common case in fuzzing is a boring trial where no novel data or code features are found. In this case, we directly ignore this test case and proceed to the next trial. Next, we refine an existing seed if the trial discovers novel data features but no novel code features. The overall idea is to reduce the number of saved seeds, since the space of data coverage is much larger than code coverage. For example, for the same static value, if the new data access reads 8 bits while the old one reads 7 bits, we directly replace the 7-bit seed with the new one. Finally, if the refinement fails to proceed, we save the current test case as a new seed and update the data-seed mapping.

Algorithm 1: Refinement-based Corpus Update

```
Input: novel code coverage C
Input: novel data coverage D
Input: code feature summaries S
Input: testcase t and code feature summary s
Data: M: maps a data feature to the saved seed
if C = ∅ ∧ D = ∅ then                          // No discovery.
    return;
if C = ∅ then
    // Data discovery only: try refinement.
    for f ∈ D do
        if S_{M_f} = s then
            ReplaceSeed(M_f, t);
            for f' ∈ D do
                M_{f'} ← M_f;
            return;
// Refinement is not possible: save a new seed.
n ← SaveSeed(t, s);
for f ∈ D do
    M_f ← n;
```

We maintain two extra data structures for efficient seed refinement. M maps a feature to the last seed saved in the corpus that demonstrates that feature. With this mapping, we can quickly enumerate the seeds for an access with the maximal access length. S maps a seed to the summary of its code features. It is used to prevent discarding already-discovered code features because of refinement. We ensure that the new

and old input shares the same code feature summary before performing the replacement.

4 Data Access Interception

We classify a program's data accesses into six categories according to their semantics. Table 2 lists an example code snippet for each category. Based on the classification, we design customized instrumentation and collection strategies to balance precision and overhead.

4.1 Integer Comparison

Integer comparisons are simple predicates. We insert a runtime library call for each integer comparison. The runtime estimates the address using the program counter. The runtime also determines the length via predicate emulation following the scheme presented in §3.1.3.

4.2 Switch Statement

Switch statements are predicates that directly determine the control flow. They are usually implemented as jump tables for efficiency [4]. Despite this, their semantics are equivalent to a series of `if-then-goto` statements using immediate values.

We emulate the semantics of switch statements accordingly. At compile time, for each switch statement we collect the case values, sort them, and store them inside a static array. We also insert a runtime library call before the switch statement, passing the condition value together with the case value array. Specifically, the function converts the switch statement into two or three integer comparisons.

If the condition value v is too large or small, we try to guide the value back to the case value range. We expand the switch as if the original code tests for `v == min` and `v == max`. In this case, a switch expands to two predicates targeted at different case values. To provide unique data coverage storage for each case, we slightly modify the program counter estimation technique when determining the address of immediate values. Specifically, for a switch executed at p, we assume the i-th case's immediate value is stored at $p+i$.

When the condition value v falls inside the range of a switch statement, we try to locate the case i such that $v \in [c_i, c_{i+1})$. Because the original case value array is sorted at compile time, we perform binary search to locate the index. If the condition value v matches the i-th case exactly, we emulate $v \neq c_i$ to reflect that the data of c_i has been completely covered. In addition to that, we also guide fuzzing towards adjacent cases, emulating $v \neq c_{i-1}$ and $v \neq c_{i+1}$ if possible. If the condition value v falls between c_i and c_{i+1}, then we emulate $v \neq c_i$ and $v \neq c_{i+1}$ to guide the condition to both cases.

For example, the switch statement in Figure 4 is located in 0x404a1c and the condition value is 14. The condition value fits between the 3rd case 4 and the 4th case 15. Therefore, we simulate $14 \neq 4$ at 0x404a1c + 3, yielding an access tuple of (0x404a1f, 8). We also simulate $14 \neq 15$ at 0x404a1c + 4, yielding an access tuple of (0x404a20, 15).

4.3 Trivial Value

Instead of instrumenting the immediate values, we rely on the existing code coverage pipeline to measure their coverage. This is because immediate values are directly embedded in instructions, and covering an immediate value implies that the parent instruction has been executed, which further implies that the basic block of the instruction has been covered. Therefore, using the existing code coverage pipeline allows us to assess the coverage of these values without introducing additional implementation complexity or execution overhead, while still maintaining completeness.

4.4 Load-Based Comparison

Load-based comparisons are a type of integer comparison, whose operator is loaded from a static value. These comparisons have the same value semantics as plain integer comparisons (see §4.1), but they have different data access semantics. To determine the access address of a load-based comparison, we must find the origin of the comparison operand.

To estimate the origin address of a value, we use intra-procedural analysis. The basic idea is to decompose the value into its dependent values recursively until we reach a load instruction. The specific rules for decomposition are shown

Table 2: Data Access Semantics and Their Address-Length Estimations

Data	Access Semantics	Example	Address Estimation	Length Estimation
Immediate Value	Integer Comparison	`x == 0`	Program Counter	Predicate Emulation
	Switch Statement	`switch (x) case 1:`	PC + Case Index	Switch Emulation
	Trivial Value	`y = x + 2`	Not Available (Represented by Code Coverage)	
Static Value	Load-Based Comparison	`p[x * y] - 1 > z`	Static Analysis	Predicate Emulation
	Region Comparison	`memcmp(p, q, 5)`	Base Pointer	Region Emulation
	Simple Load	`*p`	Pointer	Data Size

in Figure 5. During the decomposition, we remove unrelated numeric operations, such as casting (`(long)short_value`), intrinsic functions (`__builtin_bswap`), and binary operators (`x + 1` and `x | 1`). Once we have decomposed the value, we treat the load target address as the origin.

```
originOf (Load ptr)              = ptr
originOf (Cast value)            = originOf value
originOf (Intrinsic value operation)
    | operation is numeric       = originOf value
    | otherwise                  = ()
originOf (BinaryOperator lhs rhs) =
  case (originOf lhs, originOf rhs) of
    (lptr, ())                  -> lptr
    ((), rptr)                  -> rptr
    (lptr, rptr)                -> lptr
originOf (Argument _)            = ()
originOf (Constant _)            = ()
originOf (Instruction _)         = ()
```

Figure 5: Rules for deriving the origin of a value.

To instrument load-based comparisons, we use the above origin analysis to assist us. For each non-immediate comparison in the form of `lhs CMP rhs`, we insert a runtime call `trace_cmp(T lhs, T rhs, T* lptr, T* rptr)`, where `lptr` and `rptr` are the origin analysis results for `lhs` and `rhs`, respectively. For example, when instrumentation the comparison `p[x * y] > z`, we insert the runtime call `trace_cmp(p[x * y] - 1, z, p + x * y, NULL)`. The runtime function filters out non-static addresses according to the design described in §3.1.1. With the addresses filtered and the length computed through predicate emulation, we can update the data coverage at the end.

4.5 Region Comparison

Algorithm 2: Maintenance for Region Comparisons

Input: base pointer x and y, length l
Data: coverage database K
Data: novel set N
Output: comparison result
$r \leftarrow \text{OriginalComparison}(x, y, l)$;
$p \leftarrow \text{EstimateAddress}(x, y)$;
if $K_p = MAX$ **then**
  return r;
if r *is saturated* **then**
  $n \leftarrow MAX$;
else
  $m \leftarrow \text{PrefixBytes}(x, y, l)$;
  $n \leftarrow 8m + \text{UsedBits}(x_m \not\equiv y_m)$;
if $n > K_p$ **then**
  $K_p \leftarrow n$;
  $N \leftarrow N \cup \{p\}$;
return r;

Region comparisons, such as `memcmp`, are a specialized type of load-based comparison. Because data accesses in these comparisons occur in the operating system's C language runtime library, they cannot be directly intercepted by instrumenting the program's load instructions. To ensure that these data accesses are not missed, we use libFuzzer's weak hook infrastructure to reimplement these utility functions.

Specifically, C runtime libraries provide default implementations of utility functions that are marked as *weak* symbols. Since *regular* symbols take precedence over weak symbols according to linkage semantics, we provide our own versions of these functions as regular symbols, replacing the default implementations in the C runtime library. The specific logic of the interception function is shown in Algorithm 2.

First, we invoke the original C runtime library's function to compute the return value quickly. Next, we try to locate the access address and remove non-static-data accesses. Based on the comparison semantics, we mark the current comparison as saturated and skip the slow emulation if possible. If not, we compute the length in two steps: the common prefix m determines the size of the equal region at the byte level quickly, while the predicate emulation determines the number of effectively used bits for the first unequal byte precisely. We combine these results to obtain the total number of effectively used bits n and update the coverage accordingly.

4.6 Simple Load

To improve the precision of coverage tracking, we try to promote individual loads to load-based comparisons when instrumenting the program. If the promotion fails, we instrument the load as-is to ensure completeness. In such cases, we simply treat the bit width of the load instruction as the access length. However, most load instructions are not based on constant data and thus do not provide any benefits when instrumented. To avoid these cases, we use static analysis to prune them: before instrumenting a load instruction, we first locate the base memory object of the address and only proceed if the pointer is based on a static value.

5 Implementation

To assess the effectiveness of our data coverage approach, we developed a new fuzzer named WingFuzz. The fuzzer is built on top of libFuzzer from LLVM 14. We added runtime support and corpus saving logic for data coverage, and implemented instrumentation and static analysis using a compiler plugin based on the LLVM infrastructure [16].

Despite the invitation for integrating data coverage into AFL++ [11], we decided to use libFuzzer for fast prototyping when writing the paper. AFL++ is a complex fuzzer that incorporates numerous incremental research efforts. A simple integration of data coverage would potentially jeopardizing the validity of AFL++'s parameters derived from over 150 fine-tune experiments conducted on FuzzBench.

6 Evaluation

We used the classic code coverage approach and submitted WingFuzz to FuzzBench [23], the evaluation service developed by Google. FuzzBench follows the golden standard of fuzzing evaluation and publishes the results for open and reproducible science. With more than 400 experiments involving 100 fuzzers and their variants on a variety of real-world programs, it has become the de facto standard for evaluating fuzzing performance in both academia and industry.

Our evaluation follows the default setup of FuzzBench. It uses 19 target programs, runs for 23 hours, and repeats each trial for 20 times. Each trial runs on a single-core instance on Google Compute Engine with 3.75GB of memory available [23]. The baseline fuzzers used in the evaluation fully replicates the original FuzzBench paper's configuration, which includes classic works from academia and the default fuzz engines from OSS-Fuzz [1].

Through our empirical evaluation, we aim to answer the following research questions:

RQ1: Can data coverage improve fuzzing performance?
RQ2: Is data coverage orthogonal to prior techniques?
RQ3: How does individual component contribute?
RQ4: What is the runtime overhead of data coverage?
RQ5: Will the extra guidance cause state explosion?

6.1 Overall Fuzzing Performance

The evaluation summary from FuzzBench is presented in Table 3. The median branch coverage of all the 20 trials is used to calculate the metrics, where the fuzzer with the highest number of explored branches receives 100 points. The "average score" column represents the average score across all benchmarks, while the "average rank" column indicates the average of a fuzzer's relative rankings of scores.

Table 3: FuzzBench Evaluation Summary

Fuzzer	Average Score	Average Rank
WingFuzz	**98.31**	3.08
AFL++ [9]	96.91	**3.06**
Honggfuzz [31]	96.26	4.12
Entropic [5]	94.71	4.03
MOPT [21]	93.10	5.79
Eclipser [8]	92.86	6.31
AFLSmart [25]	92.84	5.28
AFL [37]	92.30	5.16
AFLFast [6]	88.77	7.70
libFuzzer [32]	87.65	7.09
LAFIntel [15]	86.47	8.17
FairFuzz [17]	84.67	7.44

- WingFuzz is based on libFuzzer.
- The full report can be found at https://www.fuzzbench.com/reports/experimental/2022-10-08-wingfuzz.

Our results demonstrate that the use of data coverage significantly enhances the average score, increasing it from 87.65 for libFuzzer to 98.31 for WingFuzz, thereby improving its rank by 9 places (12 in total). Furthermore, WingFuzz achieved the 2nd-lowest average rank (3.08), just 0.02 points behind AFL++, a sophisticated fuzzer that combines multiple incremental optimizations from the research community. An analysis of the individual benchmark ranks indicates that WingFuzz is resilient against various program types, with the minimum worst ranking of 8, while AFL++'s worst ranking was 10. Additionally, WingFuzz consistently delivers impressive results and had the lowest standard deviation (3.17) among all the fuzzers we evaluated.

To understand how data coverage assists code coverage guided fuzzer on individual programs, we compare the raw coverage improvements of individual programs between WingFuzz and its baseline version, libFuzzer. We present the raw data in Table 4, where the rows are sorted according to the gains in data coverage.

Table 4: Performance of WingFuzz Compared to libFuzzer

Benchmark	libFuzzer 23h Cov.	WingFuzz 23h Cov.	WingFuzz's Cov. Gains	Hours to 23h Cov.
sqlite3	11449.5	18142.0	+58.45%	0.25
freetype2	8173.5	12019.5	+47.05%	1.00
libxslt	8526.0	11446.0	+34.25%	0.25
woff2	987.0	1155.0	+17.02%	0.25
openthread	2247.0	2585.0	+15.04%	7.75
curl	8185.5	9046.5	+10.52%	0.25
libjpeg	1865.5	2046.5	+9.70%	1.00
libxml2	7885.0	8548.5	+8.41%	2.25
harfbuzz	7206.5	7760.0	+7.68%	1.00
mbedtls	2367.5	2527.5	+6.76%	0.50
php	15880.5	16785.5	+5.70%	0.25
proj4	4360.0	4512.0	+3.49%	4.00
bloaty	5723.0	5766.5	+0.76%	19.75
jsoncpp	519.0	518.0	-0.19%	/
re2	2566.0	2555.0	-0.43%	/
zlib	463.0	461.0	-0.43%	/
Average			+13.99%	2.96

We observe that data coverage significantly improves the coverage of a majority of programs by an average of 14%. In addition to improving code coverage, data coverage also accelerates fuzzing and reduces the time cost. On average, WingFuzz only requires 2.96 hours to achieve the same level of coverage as libFuzzer, which takes 23 hours.

In particular, data coverage performs exceptionally well on programs with large chunks of static data. For example, SQLite3 uses a static value-based table to encode parsing rules, freetype2 contains a complicated string processors for Type1 fonts, and libxslt consists of XML parsing logic. Take libxslt for a closer look. The semantics of input data are determined by predefined keys such as `<?xml`, `version`, and

`encoding`, providing more opportunities for our data coverage techniques to enhance fuzzing efficiency. However, we also observe minor regressions (-0.35% on average) on small, algorithm-oriented programs like zlib. For zlib, despite its complex in-memory operations, it uses straightforward constants such as 0 and 1 to maintain efficiency in its compression algorithms, which limits the effectiveness of advanced data coverage strategies.

> Data coverage significantly improves the overall fuzzing performance. It improved the code coverage of libFuzzer by 14%, lifting it to the 1st place in coverage score and 2nd place in average rankings. It also consistently delivers good results and has the lowest standard deviation.

6.2 Orthogonality to Prior Techniques

One valid concern about data coverage is that it could be similar to existing data-sensitive fuzzing techniques. For example, AFL++ [9] is a state-of-the-art fuzzer which constantly delivers superb performance on FuzzBench because it combines dozens of incremental research efforts including intelligent branch solving. In this section, we use AFL++ to showcase how data coverage differs from data-sensitive techniques.

First, we discovered that WingFuzz and AFL++ covered different sets of the same program in fuzzing. We assessed the raw coverage data from FuzzBench, comparing the differential "extra coverage" offered by both WingFuzz and AFL++ against the "baseline coverage" of libFuzzer across all programs. As Figure 6 shows, of all the additionally covered branches, 3,171 were detected by both, whereas 1,636 branches were exclusively uncovered by WingFuzz and 484 by AFL++. This indicates that over 34% of the additional coverage achieved by WingFuzz is unique and not replicated by AFL++, showcasing its distinctive advantage in tracking data coverage. In comparison, WingFuzz's data coverage methodology captures about 77% of the advancements AFL++ has over libFuzzer, leaving AFL++'s complex optimizations accounting for the remaining 23%.

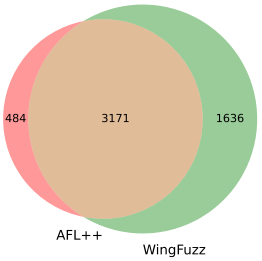

Figure 6: Extra coverage of WingFuzz and AFL++ over the baseline libFuzzer.

For a better understanding of the orthogonality between WingFuzz and AFL++, we conducted an ensemble fuzzing experiment, using the 20-hour corpus snapshot from FuzzBench for AFL++. We compared the performance of WingFuzz and AFL++ on three large benchmarks: freetype2, php, and libxslt, since these benchmarks are among the largest programs in FuzzBench, with more than 10,000 branches each. They allow for a meaningful comparison of the tools' ability to explore new code paths. We conducted fuzzing sessions for both tools on each benchmark, running for a duration of 3 hours and measured the number of unique branches discovered by each tool. The results show that WingFuzz significantly outperformed AFL++, discovering an average of 133 additional branches per benchmark compared to AFL++'s 16 (specifically, 35 v.s. 69 for freetype2, 8 v.s. 172 for php, and 6 v.s. 159 for libxslt) In other words, WingFuzz is 7x more efficient than AFL++ when fuzzing a saturated program.

Second, we found that WingFuzz can discover previously-unknown bugs in well-fuzzed programs, including AFL++. For instance, we identified two new bugs (issue #373 and #374) in the Little-CMS color management library. It's worth noting that Little CMS is a well-used package, which is integrated into widely-used software like Chromium, Firefox, and OpenJDK. These bugs eluded detection for an astonishing 14 years, even with Google OSS-Fuzz's continuous fuzzing with AFL++ and a variety of other fuzzers since 2016 and a manual audit in 2018 (refer to Issue #171). As demonstrated in Figure 7, the bug in Little-CMS pertain to its handling of ANSI/CGATS.17, a highly-structured color specification language that employs key-value pairs extensively. We adhered to established responsible disclosure guidelines: after assessing the impact of vulnerabilities, we notified the stakeholders and collaborated on mitigation to avoid misuse.

```
// BUG happens here.
nSamples = satoi(cmsIT8GetProperty(it8, "NUMBER_OF_FIELDS"));

const char* cmsIT8GetProperty(cmsHANDLE hIT8, const char* Key) {
  IsAvailableOnList(GetTable(it8)->HeaderList, Key, ...)
}

bool IsAvailableOnList(KEYVALUE* p, const char* Key, ...) {
  // Enumerates the input data by walking a linked list.
  for (; p != NULL; p = p->Next) {
    // String comparison, see Figure 2.
    if (cmsstrcasecmp(Key, p->Keyword) == 0) break;
  }
}
```

(a) Buggy code: linked list walk using static data.

NUMBER_OF_FIELDS 4 S BEGIN_DATA_FORMAT	I R G B END_DATA_FORMAT I NUMBER_OF_FIELDS 8

(b) Bug-triggering input with multiple key-value records.

Figure 7: Case study: previously-unknown bug of Little-CMS.

The library's parsing code heavily relies on data structures, featuring a linked-list walk and manually-implemented string comparison. This very characteristic facilitated the unearthing

of these bugs using our proposed data coverage approach. On the contrast, conventional fuzzers usually get lost in the same branch for different constant data references.

To further analyze how data coverage works on a variety of program types, we performed fuzzing on Serenity OS, a popular project on GitHub with over 24,500 stars. Google's OSS-Fuzz cluster continuously fuzzed it for more than two years [14], and more than 140 bugs have been discovered by code coverage-based fuzzers [24]. We conducted a bug-finding experiment on it, ensuring consistent compiler settings (`O2`, `fsanitize=address`), using default initial seeds from OSS-Fuzz, and setting an equal duration of 23 hours. The results underscore the efficacy of data coverage: WingFuzz successfully discovered 26 bugs in 17 components in the newest version of the OS (commit `4f496e97` on 2023-03-20), as depicted in Table 5. In contrast, AFL++ and libFuzzer detected 4 and 0 bugs, respectively.

Table 5: New Bugs in Serenity OS Detected by WingFuzz

Kind	Issue ID
Assertion failure	17936, 17938, 17939, 18303, 18305–18307, 18310, 18316–18321
Resource exhaustion	17937, 18309, 18312–18315
Memory safety	18036, 18044, 18302, 18304, 18324–25
Total	26 bugs

Data coverage is different from state-of-the-art techniques optimized for solving branches. For example, over 34% of the additional coverage achieved by WingFuzz is unique and not replicated by AFL++. It also found 28 previously-unknown bugs on programs well-tested by OSS Fuzz.

6.3 Contribution of Data Guidance Types

To further understand the impact of each guidance type, we conducted a controlled study using WingFuzz variants with varying levels of instrumentation. As Table 6 lists, we start with a minimal level of guidance (WingFuzz-Imm and WingFuzz-Static), and gradually restore the removed guidance until it matches the full version of WingFuzz.

Table 6: Fuzzing Guidance of WingFuzz Variants

Variant	Data Guidance		Code Guidance		Coverage
	Static	Immediate	Spatial	Temporal	
WingFuzz-Imm		Partial			60.78
WingFuzz-Static	✓				71.91
WingFuzz-Data$^-$	✓	Partial			79.25
WingFuzz-Data$^+$	✓	✓	✓		93.73
WingFuzz	✓	✓	✓	✓	94.48

One issue we encountered is related to trivial immediate values. As stated in §4.3, we use code coverage as a proxy of data coverage on trivial immediate values. However, this approach introduces code coverage, which may not be desired. Therefore, for strict control of variables, we partially collect the immediate values for WingFuzz-Imm, removing the dependency of code coverage. Similarly, we design two versions of WingFuzz-Data: WingFuzz-Data$^-$ is free from code coverage but may result in reduced data coverage of trivial values; WingFuzz-Data$^+$ is collects complete data coverage, but uses code coverage. Additionally, we disable temporal code coverage for WingFuzz-Data$^+$ to minimize the amount of information used.

We followed the evaluation best practices [13] and ran multiple 24-hour trials for 10+ benchmark programs. For unified measurement throughout this paper, we used the median value of all trials as the basis and computed the normalized coverage of each variant and presented the data in Figure 8.

WingFuzz-Imm and WingFuzz-Static only use immediate or static values for fuzzing guidance. WingFuzz-Imm only discovered an average of 61% of the maximal known coverage among all trials (14% – 84%, std = 20%). WingFuzz-Static discovered 72% coverage (39% – 92%, std = 18%), which is 11% more than WingFuzz-Imm. In fact, among all the 14 evaluated benchmarks, WingFuzz-Static outperforms WingFuzz-Imm

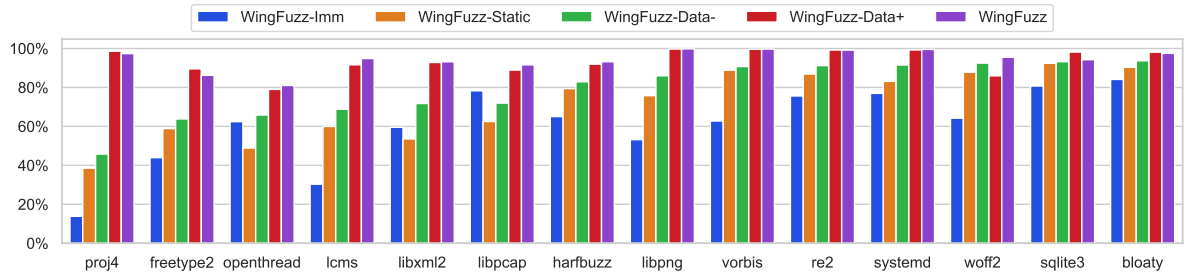

Figure 8: Normalized coverage of WingFuzz variants (see Table 6). Higher bars indicate better performance. The baseline fuzzer WingFuzz-{Static,Imm} only has static/immediate value as fuzzing guidance. The remaining fuzzers gradually enable immediate value (partial) and spacial/temporal feature of code coverage, respectively. The X axis is sorted by WingFuzz-Data$^-$.

on 11 benchmarks. This indicates that static values have rich semantics and can reflect a program's code behavior to some extent. For instance, SQLite3 is an embedded SQL engine that parses SQL using static-value-based automatons. It has 1,958 entries in the `yy_action` table. With pure static value coverage, we successfully explored 1,728 entries. With a high coverage of 88% for the action table, we obtained a high score of 92% for the overall program coverage.

WingFuzz-Data$^-$ offers a more complete view of data coverage, enabling partial immediate value guidance. Unlike static values, immediate values only carry simple semantics and are mostly trivial values. However, this scheme lacks the accounting for trivial values, which leads to a modest improvement of the coverage score to 79% (46% – 94%, std = 15%). Despite this limitation, the gains are consistent across all programs, with improvements ranging from 1% to 18%.

WingFuzz-Data$^+$ is the complete data coverage. Compared to WingFuzz-Data$^-$, it also detects trivial value usages, but introduces code coverage as a side effect. It further improves the score from 79% to 94% (79% – 100%, std = 6%). From the perspective of data coverage, the improvement can be explained by the characteristics of trivial values. Trivial values are the most common among all the three types of immediate values. For example, proj4 is a library for geodetic coordinate conversions. The computation involves a large number of trivial immediate values, such as the diameter of the Earth. When accounting for trivial values, the coverage score increased from 46% to 99%.

Surprisingly, WingFuzz-Data$^+$ outperforms the original WingFuzz on five programs, including sqlite3, freetype2, proj4, bloaty, and re2 (as shown in Figure 8). Unlike the original WingFuzz, which uses edge frequency as a guidance signal, WingFuzz-Data$^+$ leverages data coverage as a fuzzing signal, resulting in a superior signal-to-noise ratio. This approach provides strong and intuitive semantics, guiding the fuzzer to explore novel data references. In contrast, edge frequency may create noise by saving too many seeds and reducing overall efficiency.

> Data coverage is effective with or without code coverage. The use of code coverage along with data coverage can result in an additional 14% coverage, while using only static data coverage can achieve 72% of the maximum coverage observed. The number can be further increased to 94% if full data coverage is collected.

6.4 Memory and Execution Overhead

One potential drawback of using data coverage in fuzzing is the instrumentation overhead. In our implementation, we do incur a one-time memory overhead of 80 MiB. This allocation includes 16 million (2^{24}) slots for data coverage, since we limit the pointers to the last 24 bits. For each slot, we use 1 byte to store the data coverage information and an additional 4 bytes for the seed map. It is important to note that the 80 MiB allocation is performed once at program startup rather than for each seed. Instead of storing the raw coverage for each seed, we utilize a global storage mechanism to detect new coverage. This approach helps us reduce memory consumption by avoiding redundant storage of coverage information for each individual seed, enabling us to evaluate using FuzzBench, which limits the memory usage to 3.75GB.

Besides memory overhead, the collection of data coverage can also slow down fuzzing, which creates a negative impact particularly for shorter trials. To measure the actual impact of fuzzing, we extracted and analyzed the FuzzBench 15-minute snapshot, which is the earliest possible snapshot provided by FuzzBench. According to the 15-minute data, we found that WingFuzz achieved the best coverage score of 97.0 and the best average rank of 2.5. For the second-best fuzzer AFL++, is had the coverage score of 96.0 and an average rank of 2.7. As for the baseline libFuzzer, it only had 87.7 points of coverage score and had an average rank of 7.2. Therefore, data coverage is highly effective in discovering new program states, making the benefits of data coverage outweigh the overhead.

To further understand the overhead introduced by instrumentation, we measure the execution time of several differently instrumented target programs. To reduce the randomness of fuzzing, we collect the seeds of a fuzz campaign and use the same set of seeds on all programs. Figure 9 decomposes the execution duration of our instrumentation scheme, normalized to the baseline, non-instrumented program.

The blue bars of Figure 9 show the overhead introduced by code coverage instrumentation. Across the 14 programs we evaluated, the median overhead was 6% (0% – 33%, std = 8%). The red bars represent the overhead of our WingFuzz instrumentation, which includes both code and data coverage. We observed a median overhead of 61% (9% – 125%, std = 75%) for this instrumentation. This means that the introduction of data coverage reduces the fuzzing throughput by 34% compared to convention code coverage-guided fuzzing. The overhead mainly comes from the instrumentation of immediate and static values. Specifically, using immediate value tracking alone (orange bars) introduces a median overhead of 15% (-5% – 70%, std = 20%), while using static value tracking alone (green bars) introduces a median overhead of 26% (0% – 130%, std = 42%).

> The overhead of data coverage does not affect end-to-end fuzzing performance, even for short trials. In fact, data coverage yielded the highest coverage score in the shortest 15-minute evaluation. However, it did result in a reduction of throughput by 34%. This decrease is primarily due to the added overhead of collecting static values, which increases the execution duration by 26%.

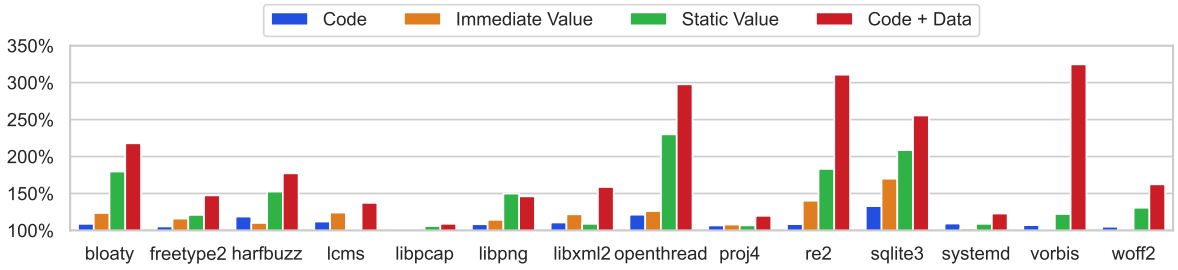

Figure 9: Normalized execution duration of programs in different instrumentation modes. Lower bars indicate better performance. For example, a value of "200%" indicates that the program executes at half the speed of the baseline version.

6.5 Queue Size and State Explosion

Fuzzers may tend to retain overly similar test cases when given very fine-grained data coverage feedback, which can hinder their ability to effectively explore and exploit the program state space within a bounded time frame. To address this issue, we employed a seed refinement strategy in WingFuzz and conducted an investigation using hourly corpus snapshots from the original FuzzBench experiment. For each duration, we normalized the queue size to the baseline of libFuzzer for each benchmark and plotted the mean value along with the error band of 95% confidence interval in Figure 10.

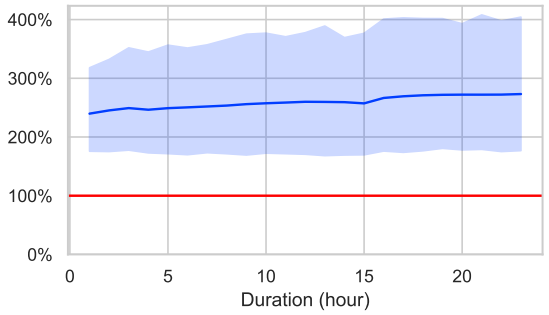

Figure 10: Temporal trend of normalized queue sizes using mean estimator with error band of 95% CI. The red horizontal line (100%) marks the queue size of libFuzzer.

Inside Figure 10, although we observed an increase in the queue size due to enhanced data coverage, it did not lead to unmanageable queue expansion. Among all trials from the FuzzBench experiment, WingFuzz discovered 139%, 150%, 159%, and 173% more seeds than libFuzzer at 1h, 6h, 12h, and 23h, respectively. This indicates a sustainable linear growth pattern rather than exponential explosion, suggesting that WingFuzz effectively balances between depth and breadth of state exploration. This analysis demonstrates the capability of WingFuzz to maintain a manageable queue size while achieving significant seed discovery rates compared to the baseline libFuzzer.

7 Related Work

Dataflow Guided Fuzzing While data coverage and dataflow-guided fuzzing share similar terminology, they represent distinct methodologies. Data coverage pertains to the analysis of a program's constant data utilization, whereas dataflow-guided approaches focus on capturing the data dependencies within code through def-use pairs.

For instance, in the context of the automaton depicted in Figure 3, where transitions are expressed as constant data, efficient exploration can be achieved through data coverage. However, distinguishing transitions becomes challenging for dataflow-guided techniques such as DDFuzz [22] and datAFLow [12]. These approaches primarily detect def-use pairs of code locations representing value definitions and usages. When the actual value is obscured behind a constant array instead of being directly associated with a code location, the differentiation of transitions becomes impractical.

Furthermore, the variance in design choices results in notable performance discrepancies. Our approach focuses solely on constant data, thereby minimizing the overhead in coverage collection to approximately 51%. In contrast, existing dataflow-guided techniques necessitate extensive instrumentation, leading to overheads as high as 10 times, as reported in datAfLow. Similarly, DDFuzz, when integrated with AFL++, exhibits decreased performance, as indicated in the FuzzBench evaluation [4].

Value Profile Introduced by libFuzzer in 2016, the "Value Profile" feature converts each comparison operation into a feature, leveraging the instruction and hamming distance between operands. Like our approach, which aims to maximize code coverage, libFuzzer endeavors to cover more value features. This technique bears some resemblance to our predicate emulation technique (see §3.1.3). However, our data coverage approach differs from Value Profile in two significant ways.

[4]The evaluation of DDFuzz and AFL++ on FuzzBench is available at https://www.fuzzbench.com/reports/experimental/2021-09-02-datadependency.

Firstly, while Value Profile diversifies discovered value features for code branches, our approach focuses on maximizing coverage for constant data. With our method, if a later execution covers more bits in a constant data operand used in the same comparison instruction, the prior seed is discarded since the newly discovered seed strictly covers more constant data than the previous one. In contrast, the Value Profile approach retains both seeds, potentially leading to seed explosion. Secondly, our data coverage approach targets constant data coverage more broadly, rather than focusing solely on immediate values used in branches. As shown in Figure 8, targeting overall static data coverage alone achieves a coverage score of 72 points, whereas the immediate-value-only variant (similar to Value Profile) only reaches 61 points.

The key distinction is that our approach aims to maximize coverage of all constant data, rather than solely solving individual branches. Interestingly, while our data coverage technique improves fuzzing performance, Value Profile actually reduces the coverage score of the original algorithm by 4 points, as reported in the original libAFL paper [10]. Indeed, among all evaluated techniques targeting fuzzing roadblocks, Value Profile performs the poorest. This broader perspective on constant data operands, rather than solely immediate values in branches, proves to be a more effective fuzzing strategy.

8 Discussion

Integration with Other Fuzzing Guidance In this paper, we delve into the concept of data coverage, presenting a novel coverage collection mechanism and scheduling strategy. While our enhancements to libFuzzer have yielded significant performance gains, the integration with other fuzzers and mechanisms is an area ripe for exploration.

Firstly, the concept of data coverage naturally aligns with more advanced code coverage forms such as context-sensitive edge coverage and branch n-grams. These approaches augment conventional code coverage by providing deeper guidance to the fuzzer. However, beyond simply introducing new guidance, it is crucial to adapt the scheduler to mitigate potential challenges, such as the seed explosion problem. For instance, when mutating a pathological input, numerous new coverage points may be generated in data coverage or n-gram code coverage. Yet, if these mutated inputs predominantly exhibit similar or identical program behaviors, blindly saving them as seeds would inundate the corpus with pathological cases, diverting resources from other potentially fruitful avenues.

Secondly, our seed replacement mechanism presents a promising solution to this challenge. Initially developed to address the seed explosion stemming from the extensive state transition table of SQLite's embedded state machine, this mechanism proved effective in curbing the proliferation of seeds, preventing out-of-memory errors caused by excessive seed generation. It remains to be seen whether this approach can be extrapolated to enhance the performance of fuzzers utilizing advanced code coverage feedback in diverse scenarios. By replacing seeds strategically, we can potentially mitigate the impact of seed explosion, enabling fuzzers to explore a broader spectrum of program behaviors efficiently.

Design of Data-Access Categories In this paper, we present an efficient coverage collection mechanism which categories the data access into six categories. While the categorization may appear detailed, it is crucial for striking the right balance between precision and practical applicability in real-world scenarios.

Our design, as outlined in Table 2, revolves around the core idea of differentiating these categories based on semantic distinctions to optimize data handling in terms of both address and length estimation. For example, if code coverage already reflects the coverage to an immediate value, we bypass instrumentation. However, for comparisons where individual bits might remain unused during execution, we employ sophisticated compile-time analysis and runtime instrumentation. This strategy minimizes operational overhead to 51%, compared to the typical 10x overhead seen in prior techniques such as datAFLow.

It is important to note that the optimal trade-off between precision and efficiency remains an ongoing exploration. The six-category design presented here is one of several possible approaches that have proven effective in practice. However, it may not be exhaustive, reflecting the design trade-offs we made between precision and efficiency. While this approach has shown efficacy within our system's context, we remain open to exploring additional data collection techniques to further enhance understanding and application.

9 Conclusion

In this paper, we explored how can data coverage assist guided fuzzing. We found that constant data encodes complex semantics that cannot be reflect by code coverage, neither can the challenge be resolved with constraint solvers. To enhance fuzzing with the guidance from constant data usages with reasonable fuzzing throughput, we designed an instrumentation based data coverage collection scheme. It also outperformed many advanced fuzzing strategies and achieved the best coverage score in the FuzzBench evaluation. We investigated the properties of data coverage with experiments. The major improvement comes from the static data, a sophisticated form of constant data with rich semantics.

Acknowledgments

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62302256, 92167101, 62021002). We

sincerely thank the anonymous reviewers for their invaluable insights and constructive feedback. Additionally, we express our appreciation to the anonymous shepherd who meticulously guided us through the refinement process, ensuring the polish and coherence of our work.

References

[1] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. google/oss-fuzz: Oss-fuzz - continuous fuzzing for open source software., 2022. https://github.com/google/oss-fuzz/.

[2] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612, 2020.

[3] 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*, 2019.

[4] Robert L. Bernstein. Producing good code for the case statement. *Software: Practice and Experience*, 15(10):1021–1024, 1985.

[5] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 678–689, New York, NY, USA, 2020.

[6] 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*, pages 1032–1043, 2016.

[7] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[8] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *Proceedings of the 41st International Conference on Software Engineering*, pages 736–747, 2019.

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

[10] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1051–1065, New York, NY, USA, 2022. Association for Computing Machinery.

[11] Van Hauser. libfuzzer is deprecated, how about porting this to afl++? · issue #1 · wingtecherthu/wingfuzz, 2023. https://github.com/WingTecherTHU/wingfuzz/issues/1.

[12] Adrian Herrera, Mathias Payer, and Antony L. Hosking. Dataflow: Toward a data-flow-guided fuzzer. *ACM Trans. Softw. Eng. Methodol.*, 32(5):132:1–132:31, 2023.

[13] 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*, page 2123–2138, 2018.

[14] David Korczynski. [serenity] initial integration by davidkorczynski · pull request #4696 · google/oss-fuzz, 2020. https://github.com/google/oss-fuzz/pull/4696.

[15] lafintel. Circumventing fuzzing roadblocks with compiler transformations, 2016. https://lafintel.wordpress.com.

[16] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization*, pages 75–88. IEEE Computer Society, 2004.

[17] 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*, pages 475–485, 2018.

[18] Michael E Lesk and Eric Schmidt. *Lex: a lexical analyzer generator*. Bell Laboratories, Murray Hill, New Jersey, 1975.

[19] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. PATA: Fuzzing with path aware taint analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1–17, 2022.

[20] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. Polar: Function code aware fuzz testing of ICS protocol. *ACM Trans. Embed. Comput. Syst.*, 18(5s):93:1–93:22, 2019.

[21] 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*, pages 1949–1966, 2019.

[22] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. Fuzzing with data dependency information. In *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy, June 6-10, 2022*, pages 286–302. IEEE, 2022.

[23] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1393–1403, 2021.

[24] OSS-Fuzz. Issues - oss-fuzz, 2023. https://bugs.chromium.org/p/oss-fuzz/issues/list?q=label%3AClusterFuzz+serenity.

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

[26] Sanjay Rawat, Vivek Jain, Ashis h Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[27] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272. ACM, 2005.

[28] Kostya Serebryany. [libfuzzer] add -trace_cmp=1 (guiding mutations based on the observed cmp instructions), 2016. https://github.com/llvm/llvm-project/commit/a5f94fb6c9cb447ebf32bef848d81ac867fd1c63.

[29] Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi, and Yu Jiang. Tardis: Coverage-guided embedded operating system fuzzing. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 41(11):4563–4574, 2022.

[30] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. Industry practice of coverage-guided enterprise linux kernel fuzzing. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 986–995. ACM, 2019.

[31] Robert Swiecki. Honggfuzz: Security oriented software fuzzer. supports evolutionary, feedback-driven fuzzing based on code coverage (sw and hw based), 2022. https://lafintel.wordpress.com.

[32] The LLVM Authors. libfuzzer – a library for coverage-guided fuzz testing., 2022. https://llvm.org/docs/LibFuzzer.html.

[33] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 61–64. ACM, 2018.

[34] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. RIFF: reduced instruction footprint for coverage-guided fuzzing. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 147–159. USENIX Association, 2021.

[35] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. Industry practice of coverage-guided enterprise-level DBMS fuzzing. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*, pages 328–337. IEEE, 2021.

[36] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[37] Michal Zalewski. American fuzzy lop, 2014. http://lcamtuf.coredump.cx/afl/.

[38] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 858–870. IEEE, 2020.