Odin: On-Demand Instrumentation with On-the-Fly Recompilation

Mingzhe Wang
Tsinghua University
Beijing, China
wmzhere@gmail.com

Jie Liang*
Tsinghua University
Beijing, China
liangjie.mailbox.cn@gmail.com

Chijin Zhou
Tsinghua University
Beijing, China
zcj18@mails.tsinghua.edu.cn

Zhiyong Wu
Tsinghua University
Beijing, China
wuzy21@mails.tsinghua.edu.cn

Xinyi Xu
Tsinghua University
Beijing, China
5354339xy@gmail.com

Yu Jiang*
Tsinghua University
Beijing, China
jiangyu198964@126.com

Abstract

Instrumentation is vital to fuzzing. It provides fuzzing directions and helps detect covert bugs, yet its overhead greatly reduces the fuzzing throughput. To reduce the overhead, compilers compromise instrumentation correctness for better optimization, or seek convoluted runtime support to remove unused probes during fuzzing.

In this paper, we propose Odin, an on-demand instrumentation framework to instrument C/C++ programs correctly and flexibly. When instrumentation requirement changes during fuzzing, Odin first locates the changed code fragment, then re-instruments, re-optimizes, and re-compiles the small fragment on-the-fly. Consequently, with a minuscule compilation overhead, the runtime overhead of unused probes is reduced. Its architecture ensures correctness in instrumentation, optimized code generation, and low latency in recompilation. Experiments show that Odin delivers the performance of compiler-based static instrumentation while retaining the flexibility of binary-based dynamic instrumentation. When applied to coverage instrumentation, Odin reduces the coverage collection overhead by 3× and 17× compared to LLVM SanitizerCoverage and DynamoRIO, respectively.

CCS Concepts: • **Software and its engineering** → **Dynamic analysis**; *Compilers*; **Software maintenance tools**.

*Yu Jiang and Jie Liang are the corresponding authors.

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523428

Keywords: Instrumentation, On-the-Fly Recompilation

ACM Reference Format:
Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. 2022. Odin: On-Demand Instrumentation with On-the-Fly Recompilation. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3519939.3523428

1 Introduction

Fuzzing is an effective approach to detect defects in C/C++ programs. The basic idea is exercising the program with random inputs. After the target program is compiled, a fuzzer generates random input, sends the input to the target program for execution, listens for bug-indicating signals as the program executes the input, and repeats this procedure continuously. Although one execution is unlikely to find a bug, a fuzzer typically executes thousands of inputs per second. The massive number of executions can effectively explore the program's logic and find bugs along the exploration. For example, over 30,000 bugs and thousands of security vulnerabilities have been found in open-source projects [1, 2]. The industry also invests in fuzzing heavily; for example, Microsoft fuzzes all its products [3, 4, 12, 13].

Instrumentation is the de facto standard for setup fuzzing in the industry [2, 12, 46]. It is vital to fuzzing in two aspects. First, some bugs, such as memory leaks and integer overflows, do not disrupt the program execution thus cannot be detected by fuzzer. With instrumentation, a bug detector embedded in the target program can abort the execution to notice the fuzzer when a bug is triggered. Second, random inputs are unlikely to trigger interesting program behaviors. To recognize and utilize these rare, high-quality inputs, guided fuzzers rely on instrumentation to provide exploration directions. For example, a coverage-guided fuzzer may collect the executed basic blocks. If the execution of an input triggers a previously-unseen basic block, then the input is saved to corpus for further exploration.

The benefits of instrumentation come at the cost of execution overhead. For example, to detect memory bugs, AddressSanitizer [47] injects verification code at each memory load and store instruction, redirects the local variables to the heap, and hijacks the heap allocator to record stack trace. It is estimated that AddressSanitizer imposes an overhead of 1.6× [57]. The high-overhead instrumentation prevails in fuzzing: 2× slowdown can be observed even on coverage collection instrumentation [54]. The overhead highly impacts the overall throughput of fuzzing, since programs are executed billions of times in a fuzzing campaign. To reduce the overhead and improve fuzzing throughput, optimizations techniques targeted at compile-time and run-time have been proposed, yet major drawbacks exist.

First, compilers compromise instrumentation correctness for speed. On the one hand, instrumenting *before* optimization harms performance. Suppose that a compiler applies AddressSanitizer instrumentation first and optimizes the instrumented program next. After instrumentation, a simple memory read can terminate the program since an assertion is inserted to check its validity. The side effect introduced by instrumentation breaks many important optimizations such as loop unrolling. On the other hand, instrumenting *after* optimization degrades correctness. This design is widely used in practice. Here the optimizations can function as usual, but the transformations performed by the optimization passes distort the fuzzing-critical semantics of the original program. With distorted semantics including values, comparisons, and control-flow graphs, instrumentation cannot detect bugs efficiently and provide useful directions for fuzzing.

Second, fuzzers rely on specialized dynamic instrumentation techniques which do not scale. For example, an already-triggered coverage probe has little contribution to a fuzz campaign, and Zeror delivers a 1.5× speed up by removing them [58]. However, the approach involves convoluted runtime support and cannot be applied to generic instrumentation schemes. Zeror modifies machine code in-memory directly via OS and hardware support. Its lightweight code patching technique ensures fast removal of probes, yet the low-level patching requires a fixed machine code layout, where instrumentation schemes commonly bloat the generated code.

The root cause of these drawbacks is the inflexibility of compiler-based static instrumentation. If a compiler can flexibly change the instrumentation scheme to fit fuzzing needs, then the aforementioned compromises become unnecessary. In other words, rather than compromising correctness for lowered overhead, removing a probe can eliminate the overhead; rather than removing the probes in a constrained way, compiler-based code emission renders convoluted runtime patching unnecessary. Despite the benefits of compiler-based static instrumentation, changing the instrumentation scheme during fuzzing is inflexible, since the recompilation can take minutes in a modern fuzzing pipeline. A new instrumentation framework is needed, so that changing instrumentation is as flexible as dynamic binary instrumentation frameworks, while the overhead is close to static instrumentation.

We propose Odin, an on-demand instrumentation framework for fast and flexible instrumentation. Odin works as an instrumentation library that cooperates with a fuzzer closely. When the fuzzer changes instrumentation requirements, Odin recompiles the program with the desired instrumentation scheme on the fly. The instrumentation change can be guided by advanced static and dynamic analysis during fuzzing. First, Odin takes the whole-program IR as the input, allowing introspecting the target program with sophisticated online static analysis. Second, the static analysis can be further enhanced by dynamic profiling results: the probe-based design of Odin enables first-class profiling support, where the profiling results can be mapped back to probes objects and annotated easily. The annotations can be `std::vector`, `llvm::DenseMap`, or even pointers to the program IR.

To accelerate recompilation, Odin first partitions the program into small code fragments before fuzzing starts. During the fuzzing campaign, when the instrumentation requirement changes, Odin locates the code fragments to change, and then re-instruments, re-optimizes, and re-compiles the small fragments. More than lowering the recompilation cost, this design additionally ensures correctness in instrumentation by moving instrumentation ahead of optimizations.

We implement the framework Odin based on LLVM [31] and evaluate it by developing a basic block instrumentation tool. Experiments show that Odin reduces the coverage collection overhead by 3× compared to SanitizerCoverage and 17× compared to DynamoRIO. Furthermore, the recompilation only takes 82 ms on average. In summary, this paper makes the following main contributions:

- We enhance on-demand instrumentation during a fuzz campaign. The enhancement features on-the-fly recompilation, which enjoys both the flexibility of dynamic instrumentation and the performance of static instrumentation.
- We implement Odin to achieve fast on-the-fly recompilation. The framework automatically recognizes the code fragments to change and limits the recompilation scope to them. Its architecture also ensures correctness in instrumentation.
- We evaluated the performance of Odin on real-world programs and observed significant performance gains over state-of-the-art static and dynamic instrumentation frameworks such as LLVM and DynamoRIO.

2 Background

2.1 Fuzzing with Instrumentation

To demonstrate how static instrumentation interacts with fuzzing, we present a case study on AFL++ [21] in Figure 1. AFL++ uses two instrumentation schemes and switches them

during fuzzing. By default, AFL++ uses the fast coverage binary to detect new code coverage with high throughput. The crude coverage information can be insufficient. For example, if the comparison used by a conditional branch is not solved by AFL++, it is desirable to know the operands of comparison to overcome the roadblock, but the coverage binary compresses this information into a boolean (whether the comparison is passed or not). In this case, the "CmpLog" binary, while being slow, can be used to collect values of the operands used in comparisons.

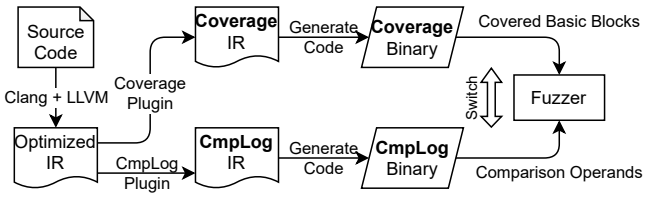

Figure 1. Instrumentation pipeline of AFL++. After lowering the source code to optimized IR, two compiler plugins instrument the program for different fuzzing feedback. The fuzzer switches the binaries during fuzzing.

AFL++ enforces a strict policy for binary switching to optimize execution speed. Because the "CmpLog" binary is slow and limits the overall fuzzing throughput, a switch only happens when an input is newly discovered and large in size. As a side effect, not all executions can benefit from the extra information. In other words, the potential of "CmpLog" binary is not maximized in the fuzzing campaign. The case of AFL++ shows that, despite the creative switch-binary approach employed by state-of-the-art fuzzers, the challenges in instrumentation still exist.

First, instrumentation correctness is compromised. As Figure 1 shows, both instrumentation plugins are invoked on optimized IR just before code generation. While optimization passes do not change the semantics concerning the program itself, they do change the semantics concerning fuzzing. For example, the "CmpLog" instrumentation is used in conjunction with the input-to-state correspondence algorithm [5] in fuzzer. The algorithm requires that the operands must be direct copies of the original input. We demonstrate that optimization can break such prerequisites in Section 2.2.

Second, both binaries are slowed down by unused probes. Besides the executed coverage probes as proven by prior works [41, 58], comparison probes also become useless once they are already solved. The fuzzing algorithm of AFL++ does not consider a comparison to be a "fuzzing roadblock" if both outcomes have been taken. The input-to-state algorithm is never invoked to solve these comparisons and these probes should be removed consequently. The removal can be achieved on the fly with dynamic binary instrumentation. However, they themselves may incur high overhead. For example, despite the help from just-in-time compilation, PIN incurs a 63% overhead without any probe installed [38].

Because the overhead in dynamic binary instrumentation can be hard to reduce, a promising option would be switching instrumentation scheme with recompilation, supposing that the recompilation of ahead-of-time compilers can be fast. To show challenges in accelerating recompilations, we present the measurement of recompilation cost in Section 2.3.

2.2 Challenges of Correct Instrumentation

The transformations performed by the optimization passes distort the semantics of the original program, affecting the bug-detection ability and fuzzing feedback quality of instrumentation. We present a case study on how instrumentation correctness is degraded in Figure 2.

```
; bool islower(char chr)
define i1 @islower(i8 %chr) {
test_lb:                          ; if (chr >= 'a')
  %cmp1 = icmp sge i8 %chr, 97
  br i1 %cmp1, label %test_ub, label %end
test_ub:                          ; if (chr <= 'z')
  %cmp2 = icmp sle i8 %chr, 122
  br label %end
end:                              ; return ...
  %r = phi i1 [ false, %test_lb ],
              [ %cmp2, %test_ub ]
  ret i1 %r
}
```

Optimize

```
define i1 @islower(i8 %chr) {
; char offset = chr - 'a';
%offset = add i8 %chr, -97
; bool r = (unsigned char)offset < 26;
%r = icmp ult i8 %offset, 26
; return r;
ret i1 %r
}
```

Figure 2. Effect of optimization on function `islower`. To check for lowercase ASCII characters, the original function contains 2 branches for testing the lower-bound `'a'` and upper-bound `'z'`, respectively. After optimization, there is only one comparison and no branching exists.

Figure 2 presents the LLVM IR before and after optimization. From the perspective of runtime performance, the optimization works exactly as expected: it folds two comparisons into one and all branches are removed. The optimization has a significant impact. For example, when simulating the generated assembly on a modern AMD Zen 3 processor, the LLVM machine code analyzer [6] reports that the optimized code is expected to be 11x faster than the original version.

However, the optimized code breaks instrumentation. As stated in Section 2.1, AFL++ uses coverage as fuzzing feedback for most cases. Without optimization, coverage instrumentation can classify the input into three categories: failed

lower-bound check (e.g. `'\0'`), failed upper-bound check (e.g. `'\255'`), and valid lowercase character. After optimization, there remains one basic block only and its feedback becomes useless: the feedback now only distinguishes whether the function has been invoked or not.

Even worse, neither can the slower "CmpLog" instrumentation work in this case. The "CmpLog" instrumentation is designed to be used with the input-to-state correspondence technique [5]. The algorithm assumes that the collected comparison operands are direct copies of the original input. Nevertheless, the optimization shifts the original value with an offset 97. For example, when the passed character is `'a'`, the value collected by `CmpLog` will be `0`. With a broken assumption, the solver algorithm cannot work anymore.

In conclusion, the optimization breaks coverage collection and value profiling of AFL++. As for systematic evaluation on impaired instrumentation correctness, we cannot enumerate them in real-world programs. Instead, we present an analysis from the perspective of optimization passes:

1. Distorted comparison semantics. The *"Instruction Combining"* pass runs the classic peephole optimization. The aforementioned case of `islower` is an example of this pass. Comparably, the *"Float to Int"* pass transforms float comparisons to integer comparisons if possible.
2. Distorted value types. The *"Basic-Block Vectorization"* and *"Loop Vectorization"* pass converts scalar values to vectors, affecting value-based instrumentation schemes.
3. Distorted control-flow graph. The loop-related passes, such as *"Loop Unroll"*, *"Loop Unroll and Jam"*, and *"Loop Unswitch"*, commit major changes to a function's control-flow graph and loop analysis results.
4. Missing/redundant basic blocks. The *"Simplify CFG"* pass can combine multiple basic blocks into one, while the *"Jump Threading"* pass can clone a basic block multiple times. The classic *"Inline"* pass also clones basic blocks, but in a bottom-up fashion along the call graph. The recursive, interprocedural optimization renders the recovery of semantics difficult if not impossible.

While optimization breaks instrumentation, a fuzzing campaign cannot afford the performance penalties of instrumenting first. The rationale is that instrumentation breaks properties of the original code, leaving less room for the following optimization passes. For example, when instrumenting memory load operations for memory-safety bugs, the inserted code aborts the program if the accessed address is invalid. The inserted code contains branches for checking the validity, and external function invocations for aborting the program. After instrumentation, the altered control flow and side effects brought by the external call cannot be recognized by the optimizer, and important optimizations such as loop unroll and loop vectorization cannot be performed.

The typical instrumentation pipeline in modern compilers is instrumenting after optimizing. Instrumenting before optimizing can provide correct and useful fuzzing feedback, but this design slows down the overall execution as a side effect. To achieve both the correctness and performance in instrumentation, recompilations should become fast enough. In such a way, the probes can be removed immediately once they become useless to fuzzing to reduce the overhead.

2.3 Challenges of Fast Recompilation

To demonstrate the cost of recompilation and directions to accelerate it, we measure the duration of each build stage in AFL++'s build pipeline [24] on target program libxml2. It is a classic target program selected by both Google fuzzer-test-suite and FuzzBench [40]. Figure 3 presents the costs of three compilation stages to build one binary. Note that AFL++ repeats the procedure twice to build two binaries with different instrumentation schemes.

The three-staged build procedure takes 40 seconds in total, which could be a substantial burden on fuzzing. In the first stage, the build system initializes for the target platform with GNU Autotools. While being usually overlooked, this stage can cost 38% of the build time. The long duration can be explained by its I/O-rich nature: enumerating various shell utilities, testing compiler option support, and checking system header files require file access and numerous compiler trial runs. In the next stage, the compiler handles each source file. This stage takes the majority of the total build time. The compiler's frontend first parses the source file and generates LLVM IR, involving file system I/O when expanding `#include` and compile-time evaluation when instantiating templates. The process can be expensive on some projects, since C++ templates are Turing complete [52]. Next, the LLVM IR is optimized, instrumented, and then lowered to Machine IR and machine code finally. Various optimization

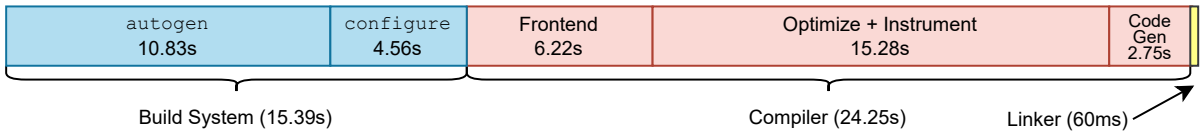

Figure 3. Breakdown of compilation cost. The initialization of the build system consists of the `autogen` and `configure` script. After the initialization, each source file is processed by the compiler's frontend to generate LLVM IR. The IR is then optimized and generated to machine code. At last, the linker produces an executable file.

passes and algorithms are involved in the process. In the last stage, the linker combines various object files into an executable. While this stage accepts all artifacts from the previous stage, this stage is relatively fast, costing a mere 0.15% of the total build time.

The breakdown of compilation cost provides valuable directions for accelerating recompilations for re-instrumentation. Since the instrumentation is performed on LLVM IR, the costly steps of build system initialization and compiler frontend can be eliminated, saving 45% of the total build time. While linking is a necessary step, it only takes 0.15% of the total build time. Therefore, accelerating the optimization and code generation step should be focused on. With a reduced recompilation scope in optimization and code generation, up to 45% of the total build time can be saved.

One prerequisite is that the reduction of recompilation scope must not impair the correctness. For example, when recompiling at basic block level, the optimal register allocation scheme can change over time. If we opt for performance and use the new scheme, the recompiled code cannot cooperate with other basic blocks correctly. The minimal translation unit of LLVM is a *module*. It is lowered to an object file after code generation. A module consists of several LLVM global values. Generally speaking, one global value maps to one symbol inside the object file. A global value can be a function, a variable, a constant, or uncommon one such as the GNU indirect function [39]. A symbol is small enough for fast recompilations. It also defines a boundary between object files with a well-defined application binary interface: an exported symbol of an object file can be imported and used by other object files. An intuitive option would be extracting the IR of the changed symbols separately and recompiling each of them. However, this approach can produce negative outcomes in three aspects:

First, there are *innate constraints* on symbol partitioning. For example, alias symbols are widely supported by multiple platforms, including ELF in [18] Linux, Mach-O [26] in OS X, and PE/COFF [28] in Windows. An alias symbol creates a different name for an existing symbol such as a function. Because relocation cannot be applied on symbols, the base symbol being aliased to must be defined rather than be declared. Consequently, the base symbol should be compiled altogether with the aliased symbol. Similarly, other limitations apply to the partition scheme, including COMDATs for supporting C++ templates and `blockaddress` expressions for C language extensions [22].

Second, *missed optimization* can occur if symbols are separated from each other. Figure 4 shows how compiler passes change the semantics and the type of symbols. Note that subsidiary text inside the presented LLVM IR is removed for brevity. Inside Figure 4, changes performed by local optimization pass *"Instruction Combining"* are marked in yellow. It replaces the library call `printf("hello\n")` with `puts("hello")`, since `puts` automatically writes a trailing new line while `printf` does not. Despite being classified as a local optimization pass, instruction combining does require access to relevant global symbols to proceed. Suppose that we reduce the recompilation scope by extracting `foo` alone, then the format string `str` becomes unavailable for inspection. Hence, the instruction combining pass cannot perform the rewrite as usual.

Third, *incorrect optimization* can occur if symbols are separated from each other. Inside Figure 4, changes of the interprocedural optimization pass *"Dead Argument Elimination"* are marked in green. Because the parameter of `foo` is unused, this pass removes the parameter inside the called function. To match the change, the corresponding argument of the caller is also removed. Not only does the parameter removal change the semantics of a function, but also the type and application binary interface. When the removal is not completed by both parties, problems arise. Suppose that we only extract function `foo` from the program, instrument the fragment first, and optimize it then. Because the caller is not present, this pass cannot remove the argument from the caller. The mismatch in type can result in program crashes. For example, the `stdcall` calling convention requires the called function to pop the argument space of the stack, and a program crashes if the stack is unbalanced. A remedy is adjusting symbol visibility. For example, the original linkage of `foo` is "internal", which means that the function is exported and thus can be solely accessed within the LLVM module. This contradicts with the fact that the already-compiled function `main` can access it. If we export the function by marking it as "external", then the dead argument elimination pass will not perform the transform since not all callers are available for modification. While the remedy ensures correctness, optimization opportunities are missed.

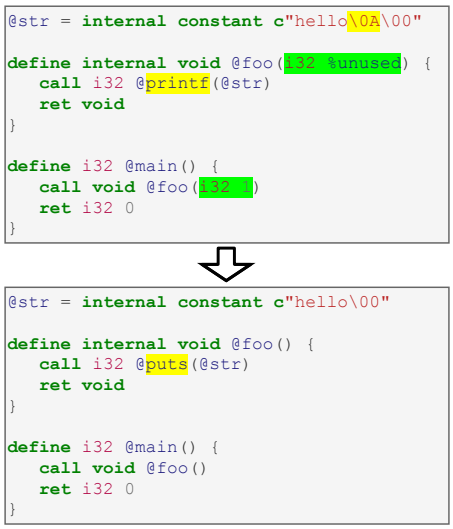

Figure 4. Effect of interprocedural optimization (marked in green) and local optimization (marked in yellow).

In conclusion, blindly recompiling individual symbols is neither correct nor fast. A fine-tuned partition scheme can fix the problems above: by recompiling related symbols altogether, the innate constraints can be fulfilled; by recompiling the symbols dependent by optimizations altogether, the missing or incorrect optimization problems can be resolved. As a side effect, the recompilation scope does increase. Therefore, the scheme should find optimal fragment sizes to balance program and recompilation performance: the fragment should be big enough for good optimization and small enough for fast recompilation.

3 Design and Implementation

3.1 System Overview

Odin strives to achieve both correctness and performance in instrumentation. Correctness is ensured by instrumenting first, where no optimizations can break fuzzing semantics; performance is improved by on-demand instrumentation, where the probes can be tuned or pruned with runtime feedback. Figure 5 presents an overview of Odin and its related components. Besides Odin, a fuzzer is added at the lower half to show how Odin closely collaborates with its user.

1. *Partition.* Before fuzzing starts, Odin creates a partition scheme for the program. The fragment definition describes the boundary between fragments. With a well-defined boundary, each fragment can cooperate with the other after lowering to machine code.
2. *Schedule.* During fuzzing, when the instrumentation requirement changes, the fuzzer modifies the probe state and requests a recompilation. The flow is emphasized with bold arrows inside Figure 5. The scheduler locates the fragments to change and creates a temporary IR encompassing all of them.
3. *Split.* After the fuzzer's patch logic completes instrumentation on the temporary IR, the instrumented IR is then split to small units according to the fragment definition.
4. Generate code. Here the LLVM infrastructure is reused for optimized code generation: each IR fragment is lowered to machine code to update the stale code cache. The cache is linked as a new executable and fuzz engine continues its operation.

3.2 Partitioning the Program

As stated in Section 2.3, the partition scheme should carefully balance the size of a fragment and ensure correctness in partition. One extreme would be creating as many fragments as possible, so that a program can be divided into small pieces for fast recompilation. The opposite extreme would be not splitting the program at all, so that all optimizations can work as usual.

To resolve the dilemma of faster recompilation versus better optimization, Odin surveys the target program to determine an optimal fragment definition before compilation starts. We present the four steps in Figure 6. For simplicity, we only assume that the interprocedural optimization pass *"Dead Argument Elimination"* removes the parameter of `neg`, and the local optimization pass "Instruction Combining" replaces `printf("hi\n")` with `puts("hi!)`.

In the first step, Odin classifies each symbol into three categories. The general idea is to summarize important constraints for optimization. If a symbol is not involved in optimization, then it can be compiled separately for fast recompilation. Otherwise, if interprocedural optimization requires a pair of symbols to function, then they are clustered together. For local optimizations, Odin clones the referenced constants locally to provide more context if possible.

1. Bond: the symbol should be defined together with other symbols, so that interprocedural optimization passes can proceed. For example, the prerequisite of removing the dead argument of `neg` is the presence of its caller `main`, since they must be modified in pairs. The trial optimization of Odin detects the dependency and clusters them altogether.

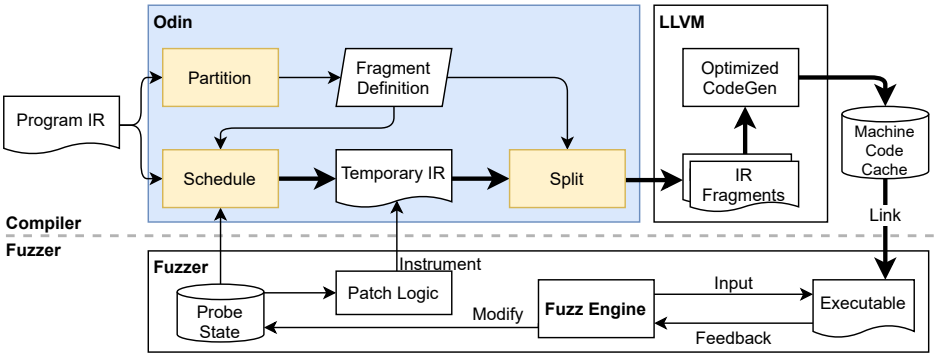

Figure 5. System design of Odin. Odin works as a library inside the fuzzer. When recompilation is requested, Odin recompiles changed code fragments on the fly with the LLVM infrastructure. The machine code cache is updated with the results, and linked to a new executable for fuzzer's use.

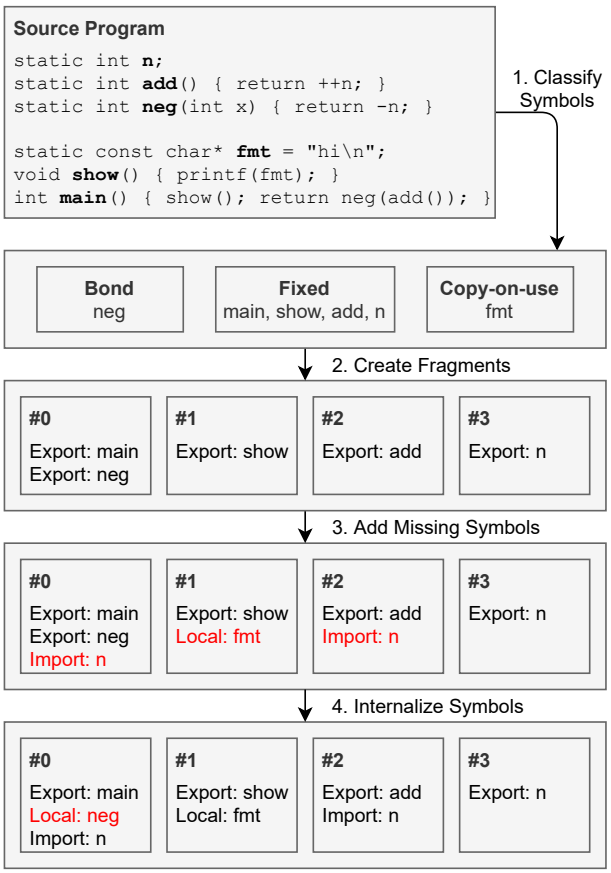

Figure 6. Steps for partitioning a program. First, ODIN classifies the symbols to summarize related constraints on optimization. Next, fragments are created for symbols with constraints. Finally, ODIN adds the missing symbols, internalizes exported symbols, and stores the fragment definition for recompilation use.

2. Copy-on-use: the symbol should be copied into a fragment when being referenced, so that local optimization has enough contexts. For example, the instruction combining optimization requires the presence of the referenced symbol `fmt` to optimize the referencing function `show`. For semantically non-clonable symbols, they are marked as "Bond" with its users. Additionally, they are marked as "internal" to prevent conflicts at link time.
3. Fixed: the symbol should be defined as-is. Fixed symbol has a stable application binary interface to enable cooperation with other code fragments at binary level. All symbols belong to this category by default. For example, the externally-used function `show` and `main` is fixed naturally; function `add` is fixed since no optimization changes its type; variable `n` belong here since it does not belong to other cases list above.

The classification is mainly based on the symbol's semantics and interprocedural optimization requirements. The semantics is derived from the IR, which covers the *innate constraints*; the requirements are collected from a trial optimization run, where the compiler passes (modified by ODIN) log the requirements for later inspection.

In the second step, ODIN creates fragments to establish coarse boundaries between groups of symbols. This step is focused on symbols with constraints. For example, as stated in Section 2.3, the *innate constraint* requires two symbols to appear altogether or the generated object file would be incorrect. Algorithm 1 shows the detailed steps. First, in line 3–11, two types of symbols are clustered together: symbols with innate constraints are clustered for correctness, and the "Bond" symbols are clustered together to allow further optimizations. In line 12–14, a new fragment is created for each cluster. Next, in line 15–20, the remaining "Fixed" symbols are also created with a fragment. Because the locally-cloned "Copy-on-use" symbols do not affect the boundary of a fragment, the addition of them is postponed to the third step.

Algorithm 1: Create Fragments

Data: LLVM IR M and category map C
Result: Fragment collection $F = \{f_1, f_2, \ldots, f_n\}$; each fragment is a set of symbols

```
1  Initialize union set U to store clustered symbols ;
2  Initialize partition F as empty ;
3  foreach Symbol p ∈ M do
4      foreach Symbol q ∈ M do
5          if p and q has innate partition constraint then
6              join(U, p, q) ;    /* Ensure correctness */
7          else if C[p] is "Bond" and q references p then
8              join(U, p, q) ;    /* Allow optimization */
9          end
10     end
11 end
12 foreach Cluster c ∈ U do
13     F ← F ∪ c;
14 end
15 foreach Symbol p ∈ M do
16     if ∀f ∈ F : p ∉ f then
17         if C[p] is "Fixed" then
18             F ← F ∪ {p};
19         end
20     end
21 end
```

In the third step, ODIN adds missing symbols to fragments. For each fragment, it scans the referenced symbols and adds the missing ones. Importing a missing symbol ensures IR correctness at recompilation time, since a well-formed IR cannot reference undefined symbols. If the missing symbol is categorized as "Copy-on-use" (e.g. `fmt` inside fragment #1),

then the missing symbol will be cloned locally at recompilation. The scan-and-add operation is performed recursively, since a cloned symbol may reference previously-unseen missing symbols. Otherwise, the symbol's semantics does not permit cloning it at recompilation time. For example, `neg` inside fragment #0 references the variable `n` which should be defined for once only. In this scenario, its declaration is imported instead.

In the last step, Odin internalizes exported symbols. Because adding missing symbols can create imports from one fragment to another fragment, all symbols are set as exported when first created. The post-processing step scans the references to the exported symbols. If no cross-fragment reference exists, then the exported symbol is internalized. For example, inside Figure 6, the symbol `neg` can only be accessed by the locally defined function. Therefore, it is marked as "local" so that the dead argument elimination pass can proceed.

The bottom row of Figure 6 presents the output of the precompile survey. We can see that all possible optimizations are preserved, yet the program is split into four fragments. Recompilation can be accelerated with a reduced scope.

3.3 Scheduling the Recompilation

When a fuzzer completes modification to the probe state, the changed instrumentation scheme can be applied to the executable via recompilation. Figure 7 presents the overall steps for recompilation.

The first step is to *schedule* the necessary fragments to recompile and probes to re-apply. Algorithm 2 presents the scheduling process in detail. The algorithm contains three stages of propagation. The rationale behind a multi-staged propagation is Odin's data model. For example, in Figure 7, the only changed patch is associated with the function `bar`, and the function is used in fragment #0 and #1. Because Odin recompiles at fragment level, the remaining function inside these fragments, i.e., `foo` and `baz`, must be recompiled altogether. Consequently, probes targeted at `foo` and `baz` also get scheduled.

In line 2–6 of Algorithm 2, Odin detects probes to change and associates them with symbols. While a probe can attach to one symbol only, a "Copy-on-use" symbol can be cloned into multiple fragments. Therefore, in the second loop (line 7–11), Odin propagates the changed symbols to fragments, since the recompilation unit of Odin is a fragment rather than a symbol. However, the propagation results in inconsistency in probes. The recompilation of a fragment affects all symbols inside the fragment. Besides symbols with changed probes, other symbols with unchanged probes inside one fragment should be instrumented, too. Therefore, in the last loop (line 13–17), Odin back propagates the fragments to probes. In the end, a temporary IR is created by duplicating all changed symbols inside the original IR (line 18). Note that the back propagation should *not* be repeated until convergence: it only adds unchanged probes, whose fragments' caches are still valid for reuse.

After instrumentation completes, Odin splits the temporary IR back to small fragments. As Figure 7 shows, two small IRs are created by extracting the related symbols in

Algorithm 2: Scheduling Fragments and Probes

Data: Probe collection P and fragment collection F
Result: Probes to apply $\tilde{P}$ and temporary IR $\tilde{M}$

1 Initialize the set of changed symbols $\tilde{S} \leftarrow \emptyset$;
2 **foreach** *Probe p, Symbol* $s \in P$ **do**
3 **if** *p is changed* **then**
4 $\tilde{S} \leftarrow \tilde{S} \cup \{s\}$;
5 **end**
6 **end**
7 **foreach** *Fragment* $f \in F$ **do**
8 **if** $f \cap \tilde{S} \neq \emptyset$ **then**
9 $\tilde{S} \leftarrow \tilde{S} \cup f$;
10 **end**
11 **end**
12 Initialize the set of probes to apply $\tilde{P} \leftarrow \emptyset$;
13 **foreach** *Probe p, Symbol* $s \in P$ **do**
14 **if** *p is activated and* $s \in \tilde{S}$ **then**
15 $\tilde{P} \leftarrow \tilde{P} \cup \{p\}$;
16 **end**
17 **end**
18 $\tilde{M} \leftarrow ExtractIR(\tilde{S})$;

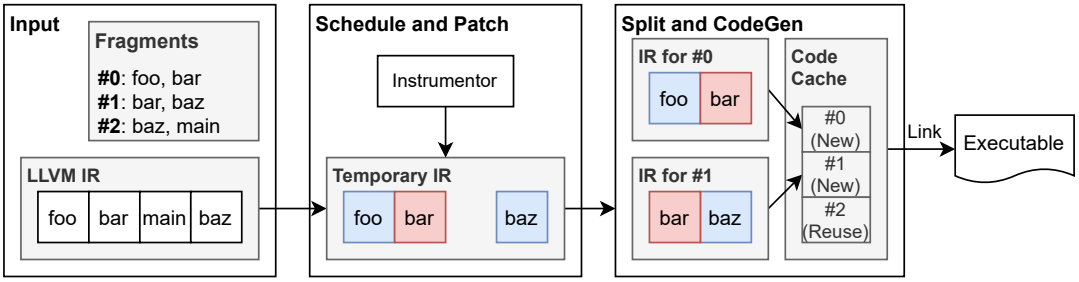

Figure 7. Steps for recompilation. This example assumes that a probe targeted at the function `bar` (marked in red) is changed. Other functions with unchanged probes are marked in blue.

fragment #0 and #1, respectively. While the copy-instrument-split design is complex to implement, it simplifies the instrumentation logic at the user's side. The user is not required to understand the concept of fragments and manually re-apply the same probe for different fragments.

Finally, for each changed fragment, the instrumented IRs are adjusted for symbol visibility, optimized with LLVM's pipeline, and lowered to machine code in the end. As Figure 7 shows, fragment #0 and #1 are recompiled and the code cache is updated accordingly. All cached machine code is then linked to an executable. Because of caching and limited scope in recompilation, this procedure is fast and can be repeated frequently within the fuzzing campaign.

4 Instrumentation with ODIN

We implement ODIN as an on-demand instrumentation framework for fuzzing. It accepts a whole-program IR as the input, performs initial cleanup, and then assists the user to split instrumentation into individual probes following the OOP paradigm. For example, to record values used in comparisons (`CmpInst` in LLVM), we can create a `CmpProbe` class, where each instance of `CmpProbe` only targets one `CmpInst`:

```
// A probe recording operands used in comparisons.
struct CmpProbe : Probe<CmpProbe> {
  // Probe-specific information can be stored here freely.
  const CmpInst *TheCmp;      // The comparison to instrument.
  uint64_t LastObservedValue; // Dynamic information from profiling.

  // The framework invokes this method to find the function to patch.
  const Function *getPatchTarget() const {
    return TheCmp->getFunction();
  }
};
```

ODIN provides `PatchManager` for dynamic adding, deleting and changing probes:

```
Probe &Old = Manager.getProbe(0); // Probes can be queried.
Manager.remove(Old);              // Probes can be removed.
std::shared_ptr<CmpProbe> NewProbe = Manager.add<CmpProbe>(Inst1);
NewProbe->Instruction = Inst2;    // Probe logic can be changed.
```

A recompilation can be scheduled when the modification to probes is completed. The `Scheduler` detects the minimum set of probes to apply, and creates a temporary copy of the original IR for patching. Patch logic developed by the user can easily map the original IR to the temporary IR when applying the probes:

```
Scheduler Sched = Manager.schedule();

for (auto *P : Sched.ActiveProbes) {
  CmpProbe *Probe = cast<CmpProbe>(P);
  // Get the temporary instruction cloned for this recompilation.
  CmpInst *TheCmp = cast<CmpInst>(Sched.map(Probe->TheCmp));

  // User logic comes here. It is similar to static instrumentation:
  // just manipulate the IR with LLVM's infrastructure.
  // In this case, we add a call to runtime function.
  IRBuilder<> IRB(TheCmp);
  Function *RuntimeFn = Sched.lookupFunction("on_cmp");
  IRB.CreateCall(RuntimeFn, {...});
}

// Optimize, generate code, and invoke the linker.
Sched.rebuild();
```

The rationale behind creating temporary IR is reverting instrumentation changes. Modifying the original IR can be easy at first, but reverting the changes is difficult. To reduce the development complexity for instrumentation authors, ODIN follows the functional approach. For each recompilation, ODIN creates a temporary copy of the original IR. The user can map the original IR to the temporary IR easily, and then instrument the temporary one.

5 Experimental Evaluation

We evaluate the performance of ODIN by measuring the execution duration of instrumented programs. We replay the seed files collected during an 24-hour fuzzing campaign. By replaying the seed files, we can avoid randomness caused by fuzzing.

For the comprehensiveness of target programs, we select every program occurring in both Google fuzzer-test-suite and FuzzBench [40]. Handpicked by Google, they encompass a typical set of real-world programs. For fairness, all the programs are built with Clang 12 with the same optimization level `O2`.

We choose basic block coverage, a simple and widely-supported instrumentation scheme. We ensure that all evaluated coverage tools use the same scheme for fairness. As such we deliberately exclude advanced instrumentation schemes, including AFL and AFL++. We evaluate SanitizerCoverage [7] of LLVM for its considerable influence on the industry — it is deployed on thousands of nodes on Google's fuzzing cluster. Since it can collect other kinds of coverage, we only enable the pure 8-bit-counter-based basic block instrumentation for fairness. We evaluate DrCov of DynamoRIO [10]. It is one of the fastest dynamic binary instrumentation frameworks. Leveraging just-in-time compilation, it can achieve up to 7.3× performance improvements [23] over PIN [38]. We evaluate libInst of the famous instrumentation framework DynInst [8]. Note that libInst only uses the static binary rewriting functionality of DynInst.

For demonstration use, we implement ODINCOV to record the hit count for each basic block and prune unused probes at runtime like Untracer [41] does. We also implement ODINCOV-NOPRUNE, a weakened version of ODINCOV *without* runtime probe pruning.

Tool	Framework	Type	Target
ODINCOV	ODIN	Dynamic	Compiler
SanitizerCoverage	LLVM	Static	Compiler
DrCov	DynamoRIO	Dynamic	Binary
libInst	DynInst	Static	Binary

5.1 Overall Performance

To demonstrate how the flexibility of on-demand instrumentation turns into performance gains, we present the execution duration of the instrumented programs over different

instrumentation tools in Figure 9 and 8. To avoid bias introduced by programs, we further normalize the metric by dividing the instrumented version's duration with the baseline version's duration. Note that libInst crashed on freetype2, re2, and json. We omit them when calculating metrics. A series of conclusions can be drawn from these figures.

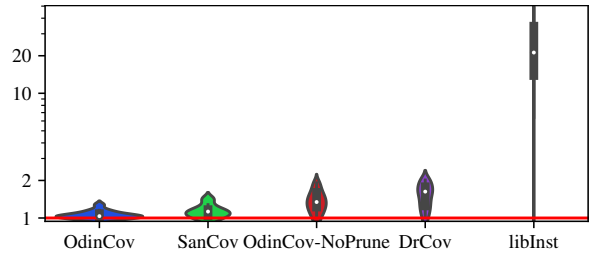

Figure 9. Normalized execution duration of all instrumented programs. The red bar marks the baseline, non-instrumented programs.

First, OdinCov achieves the lowest overhead among all coverage tools. The median overhead for OdinCov is a bare 3.48%, while the median overhead is 15% for SanitizerCoverage, 63% for DrCov, and 1,920% for libInst. In other words, the coverage collection performance of OdinCov is 3× better than SanitizerCoverage, 17× better than DrCov, and 551× better than libInst. Odin can easily remove the redundant probes as fuzzing makes progress. Consequently, the programs instrumented with OdinCov only keep the useful coverage probes and an extremely low overhead is achieved. Furthermore, the architecture design of Odin allows runtime introspection of a program IR at fuzzer's side, which enables more complex probe designs beyond basic block coverage.

Second, Odin promotes correct instrumentation without compromising speed. When evaluating static instrumentation performance by disabling probe pruning on purpose, OdinCov-NoPrune is slower than SanitizerCoverage by 23% on average. The reason behind the cost is different design choices: as an industry-standard instrumentation tool, SanitizerCoverage compromises instrumentation correctness for speed. The pass is placed at the very end of the optimization pipeline, since early instrumentation may break optimizations. On the contrary, the on-demand instrumentation provided by Odin dismisses the concerns of probe performance, allowing instrumentation to be put in advance for correctness. For example, in this case of block coverage instrumentation, OdinCov improves the performance of OdinCov-NoPrune by 22% on average.

Third, Odin allows easy and fast dynamic instrumentation. Besides Odin, dynamic binary instrumentation frameworks also allow changing the instrumentation scheme on-the-fly. While the binary-based instrumentation approach enables flexible, source-free coverage collection, the performance can be limited by the lowered representation of the program, even with just-in-time binary translation. For example, the programs instrumented by DrCov are slower than OdinCov-NoPrune by 16% on average. As for DynInst-based tool libInst, the slow down is 16× on average. Because Odin takes the LLVM-based approach, comprehensive analysis and optimization passes of the LLVM infrastructure can be reused to improve the code quality. This approach also allows easier inspection of the program: rather than relying on callbacks on various events, Odin allows inspecting the whole program at the fuzzer's side. The natural approach can be much easier to program. For example, the probe setup, instrumentation, and prune logic for OdinCov takes 33 lines of code in total, while the main file of DrCov [27] registers 7 kinds of callbacks and takes ~600 lines. Furthermore, LLVM IR also provides richer semantics than the lowered machine code. For example, a jump table at binary level resembles a soup of code pointers, while it can be a well-defined `switch` instruction in IR. The rich information can be used to lower instrumentation overhead and provide directions in fuzzing.

5.2 Overhead of Recompiled Programs

To demonstrate that Odin produces high-quality code, we evaluate the execution duration on compiled programs. Since the evaluation is not related to specific instrumentation schemes, we disable instrumentation here and compile the program as-is. To control the variables, we develop two alternative versions as Table 1 shows: Odin-OnePartition does

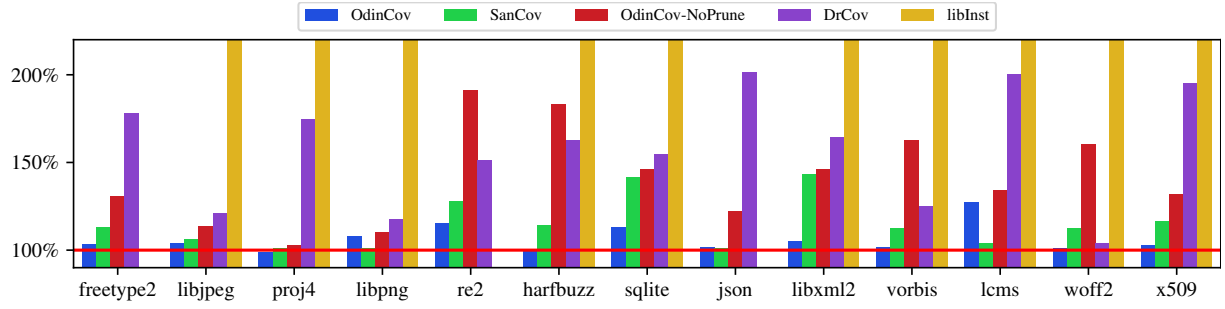

Figure 8. Normalized execution duration of instrumented programs. The X axis is programs, the Y axis is the normalized execution duration to the corresponding non-instrumented programs. Lower bars indicate better performance.

not partition the program at all and provides more optimization opportunities; Odin-MaxPartition tries to create as many code fragments as possible, lowering the scope of recompilation as a result.

Table 1. Variants of Partition Schemes

Variant	Code Fragments	Feature
Odin	(Original)	
Odin-OnePartition	1	Better Optimization
Odin-MaxPartition	Max Possible	Faster Recompilation

Figure 10 presents the execution duration of Odin variants. Compared with the baseline programs, the overhead of Odin-OnePartition, Odin, and Odin-MaxPartition averages 1.12%, 1.43%, and 55.77%, respectively.

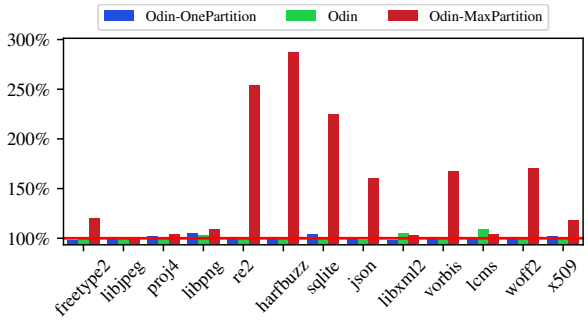

Figure 10. Program execution duration of non-instrumented programs of partition variants. For each program on each partition variant, the collected execution duration is normalized to the execution duration of the compiler's original, non-instrumented output (marked as the red bar).

The bars of Odin-MaxPartition show that, blindly partitioning the program negatively impacts the performance on a number of programs. The overhead fluctuates for different programs: among all the 13 programs evaluated, 6 programs have an overhead over 50%. The overhead of the worst-performing program harfbuzz is 186.91%; while for the best-performing program, libjpeg, it incurs an overhead of 0.95%. The reason can be explained by reduced optimization interprocedural opportunities: the compiler can only access the code fragment being compiled (one function in most cases); therefore, the blind partition scheme reduces the overall performance, especially for programs relying on interprocedural optimizations.

On the contrary, the partition of Odin takes the requirements of optimization into account. By surveying the program with a trial optimization, it partitions the program in a manner where the necessary context for optimization is preserved. For this reason, the slow down of Odin is a bare 0.31% compared to the non-partitioning variant.

5.3 Overhead of On-the-Fly Recompilation

One concern of Odin could be the extra recompilation cost: compared to Odin-MaxPartition, Odin places more code inside one fragment. Consequently, Odin may spend much more time recompiling the fragment. To demonstrate the partition scheme of Odin can also achieve fast recompilations, we present statistics on recompilation time of code fragments in Figure 11 and 12. Note that the time spent in build systems is excluded for fairness: we assume that the cost can be reduced by caching the LLVM bitcode produced by compiler front ends. Consequently, we only measure the total time spent in the compiler's middle end and back end.

Figure 11 presents the average recompilation time to compile each fragment. The compilation time is normalized with Odin-OnePartition. Its one-partition design recompiles the whole program and saves no time. Compared with the best-possible partition scheme Odin-OnePartition, Odin saves 97.91% of the recompilation time on average. The saved time can be explained by the approach that Odin follows: rather than recompiling the whole program, Odin partitions the program and only recompiles the changed fragment. For most programs, only a small chunk of code is necessary for optimization passes to function or should be clustered together. Therefore, the scope for recompilations becomes smaller as the program's size grows. For example, the average compilation time for the simple header-only parser "json" is 3.63% of the original time, while the time for the complex program "sqlite", an embedded SQL engine, is 0.09%. In other words, Odin can be scaled to large-scale programs.

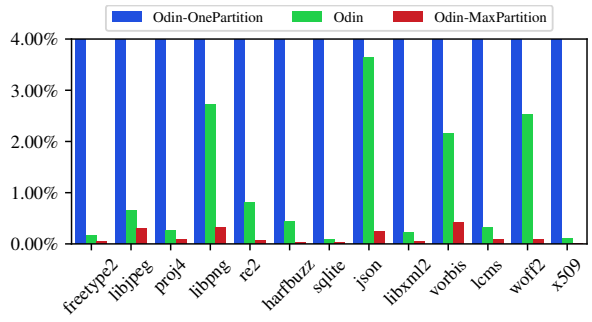

Figure 11. Average recompilation duration of all code fragments. The metric is normalized to the total time of recompiling the whole program.

Compared with the fastest-possible partition scheme Odin-MaxPartition, Odin spends 6.5× more recompilation time on average. The reason for extra time is that Odin expands the recompilation scope to allow optimization passes to function. Take json, a header-only C++ template library for example. Its extensive use of C++ templates results in short functions suitable for interprocedural optimization. Among all the 544 functions defined in it, only 27 functions are generated

to code while any other functions are removed after optimization. To preserve optimization opportunities, Odin only splits the program to a few dozen fragments, rather than hundreds of fragments as Odin-MaxPartition. Consequently, Odin-MaxPartition takes 2.03 ms to compile a fragment on average, while Odin takes 30.67 ms. With the minor difference in recompilation cost, Odin can optimize the program better and outperforms Odin-MaxPartition by 63% with regard to program execution duration.

Beyond analyzing the average performance gains of recompilations, we further show that Odin is robust against the worst cases. This situation can happen if the code fragment being recompiled is too large, or the code being compiled triggers an expensive optimization algorithm. Figure 12 presents the absolute duration to recompile the slowest fragment. The median time of recompiling the slowest fragment is 542 ms, and only three programs' worst-cast recompilation time exceeds 1 second.

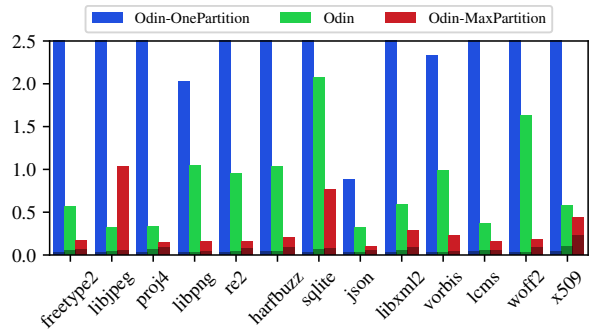

Figure 12. Worst-case re-instrumentation duration (in seconds). The dark bars below are for linking, and the bright bars above are for recompilation.

Take the worst-case "sqlite" for example. As an embedded SQL engine, SQLite places all SQL execution logic inside the function `sqlite3VdbeExec`. The complexity of SQL leads to this enormous function: it counts 6,475 lines in source code, handles the execution of 163 opcodes, compiles to 2,058 basic blocks, and references 156 external symbols. It is no wonder that the compilation can be costly even for the fastest-possible scheme Odin-MaxPartition, which takes 0.69 seconds to compile this fragment. While Odin spends 1.3 more seconds in recompilation compared to Odin-MaxPartition, it ensures the quality of generated code: as Figure 10 illustrates, Odin only incurs an overhead of 0.3% compared to the baseline program, while for Odin-MaxPartition which compiles faster, an overhead of 125% is observed.

Figure 12 also presents the cost of linking the whole program, The linking only averages to 49 ms because it does not involve much computation compared to an optimizing compiler. Furthermore, Odin only exports few symbols because of internalization of fragments. Consequently, the linker has less symbols to resolve compared to a normal build.

6 Related Work

6.1 Use of Instrumentation in Fuzzing

Instrumentation helps to provide exploration directions in fuzzing. Because fuzzing is a dynamic testing approach, triggering the code is the prerequisite to detect bugs in it. Besides enhancements on plain code coverage [36, 55], a variety of extra information is used for guide fuzzing, including call stack [43], branch comparisons [34, 37], and memory access patterns [56].

Instrumentation also helps to detect bugs in fuzzing. By reporting a bug, the most trivial bugs such as integer overflow can be detected by fuzzing. The sanitizers are a family of compiler-based bug detectors widely-used in production [50]. They can detect common C/C++ bugs including thread races [48], buffer overflow [35, 47], uninitialized memory [51], and type confusions [46]. There also exists bug detectors for binaries, for example, DynamoRIO-based DrMemory [11] and Valgrind-based MemCheck [49].

These instrumentation schemes greatly improve the bugs detected of the overall fuzzing campaign, yet their overhead limits the throughput of execution. By switching the instrumentation scheme with Odin, the throughput can be increased significantly and the computation cost can be reduced consequently.

6.2 Reducing Instrumentation Overhead

Instrument less and the overhead reduces naturally. For each probe, the cost can be reduced with compile-time optimization [54] and subtle data structures [16, 29]. Among all probes, the unnecessary ones can be detected with graph theory [25], profiling [33, 53], and hybrid analysis [57]. Recent works [17, 45] also accelerate fuzzing with hardware features such as Intel Processor Tracing. For example, JPortal [59] preserves the accuracy for end-to-end Java bytecode control flow profiling while achieving an ultra-low overhead as low as 4%. The works above are extremely effective for the proposed instrumentation scheme. However, the removal of unnecessary probes is a major challenge in these works.

On the one hand, runtime patching is inflexible. To avoid disrupting fuzzing, researchers resort to lightweight binary rewriting [41] and code patching [58]. However, it is difficult if not impossible to apply these fine-tuned approaches to other instrumentation schemes. For example, because runtime binary rewriting with DynInst is expensive, Untracer [41] caches the code layout before fuzzing starts and rewrites the binary directly during fuzzing. All aforementioned works require the same code layout, while instrumentation schemes commonly bloat the generated code.

On the other hand, recompilation is expensive. Because a compilation can take minutes, all aforementioned works perform their optimizations before fuzzing starts, losing the chance to refine the probes inside a fuzzing campaign. For

example, ASAP [53] observes that frequently-triggered sanitizer checks are unlikely to detect bugs. Because it is difficult to remove the probes during fuzzing, ASAP builds an additional profiling binary to collect the cost of each check, then removes these checks on the next build before fuzzing starts.

Because Odin regenerated the code rather than patching it directly, it can be applied to a variety of instrumentation schemes. Furthermore, as stated in Section 5.3, the recompilation costs a bare 82 ms on average. The low cost permits frequent refining of the probes.

6.3 Dynamic Binary Instrumentation

Binary instrumentation frameworks modify a program at binary level before [32] or during execution [38]. Various techniques are used to optimize the performance and enhance the capabilities.

Lightweight binary instrumentation frameworks (LDBI) are cheap per se. Systems like XRay [9] in LLVM and ftrace in Linux kernel inserts no-op slides at potential probe points. When a probe is activated, the corresponding no-op slide is overwritten with jumps to trampolines to redirect the control. WordPatch [14] and LiteInst [15] further reduce the cost of inserting no-op slides. However, only a handful of instrumentation schemes is applicable to lightweight instrumentation, because they require static analysis and code relocation features, which are missing from LDBI. First, LDBI cannot provide many basic analyses because the machine code is low-level. They cannot even reconstruct the basic blocks inside a function, not to mention recognizing the probe code injected by instrumentation. Second, LDBI cannot change the layout of machine code. To remove the probe code from the application code, LDBI solutions can only replace the instructions with "nop"s. These nop instructions still burn CPU cycles and slow down the execution.

Heavyweight instrumentation frameworks are more capable. More than redirecting the execution, they usually provide primitive analysis functionalities to inspect the program. For example, DynamoRIO [10] and PIN [38] allow inspection and modification to raw instructions; Valgrind [42] abstracts away platform differences with its VEX IR; DynInst even allows high-level inspection at module, function, basic block, and instruction levels. However, the code at machine level is not easy to analyze or patch. Overhead up to dozens of times are observed [42].

Odin operates on the high-level LLVM IR. The design choice brings two benefits. First, analyzing and manipulating the SSA-based LLVM IR is easy, especially when compared to raw instructions or VEX IR with side effects. Second, LLVM IR encodes high-level information unavailable in binary form, which can be used to develop sophisticated instrumentation schemes. Finally, the infrastructure of LLVM enables optimized code generation. As stated in Section 5.1, Odin is 16% slower than the weakened version of Odin, which has dynamic probe pruning disabled.

6.4 Compiler Cache

Compiler cache accelerates recompilation by only invoking the compiler on changed source files. For example, Ccache [44] and sccache [19] first compute the hash of dependent input files and cache the result; if the following recompilation requests share the same hash, the cached result is directly used. Recent works further refine the granularity from files to AST [20, 30].

While Odin leverages recompilation, it is not a standalone recompilation cache tool which invokes the compiler lazily. It works as an evolutionary instrumentation library working with fuzzers closely: more than simply generating object files, it collects profiles at runtime, updates individual probes' statistics, and adjusts the binary accordingly. The process is repeated as fuzzing makes progress.

7 Future Work

The application of Odin is not limited to reducing the overhead of code coverage. Take ASan as an example, ASAP [53] observes that bugs are commonly located in cold checks; to reduce the overhead of hot checks, ASAP first profiles to locate the hot checks and then removes them with a rebuild. This approach, while being effective, cannot remove the hot checks not yet covered in the profiling run. With Odin, hot checks discovered in fuzzing can also be removed. Take UBSan for another example. Because of its high false-positive rate, most programs terminate even on well-formed inputs. With Odin, UBSan can be used with fuzzing easily: a faulty probe can be removed immediately once triggered, allowing the whole fuzz campaign to continue.

8 Conclusion

In this paper, we propose Odin to accelerate instrumentation in fuzzing. It follows the on-demand instrumentation approach to lower the programs' overhead and leverages on-the-fly recompilation to reduce the recompilation cost. The approach achieves the flexibility of dynamic instrumentation while preserving the performance of static instrumentation. As an example of coverage instrumentation, Odin's coverage collection overhead is 3× lower than SanitizerCoverage and 17× lower than DynamoRIO. Furthermore, the recompilation only takes 82 ms on average.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. Our shepherd, Zhiqiang Zuo, also provided helpful feedback, for which we greatly appreciate. This research is sponsored in part by the NSFC Program (No. 62022046, 92167101, U1911401, 62021002, 62192730), National Key Research and Development Project (No. 2019YFB1706200, No. 2021QY0604).

References

[1] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. 2016. Continuous fuzzing for open source software. https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html. [Online; accessed 15-May-2021].

[2] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. 2016. GitHub - google/oss-fuzz: OSS-Fuzz - continuous fuzzing for open source software. https://github.com/google/oss-fuzz. [Online; accessed 04-Nov-2021].

[3] Abhishek Arya, Oliver Chang, Max Moroz, Martin Barbella, and Jonathan Metzman. 2019. GitHub - google/clusterfuzz: Scalable fuzzing infrastructure. https://github.com/google/clusterfuzz. [Online; accessed 04-Nov-2021].

[4] Abhishek Arya, Oliver Chang, Max Moroz, Martin Barbella, and Jonathan Metzman. 2019. Open sourcing ClusterFuzz. https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html. [Online; accessed 15-May-2021].

[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. Redqueen: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[6] The LLVM authors. 2021. llvm-mca - LLVM Machine Code Analyzer — LLVM 13 documentation. https://llvm.org/docs/CommandGuide/llvm-mca.html. [Online; accessed 01-Nov-2021].

[7] The LLVM authors. 2021. SanitizerCoverage — Clang 13 documentation. https://clang.llvm.org/docs/SanitizerCoverage.html. [Online; accessed 09-Nov-2021].

[8] Andrew R. Bernat and Barton P. Miller. 2011. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. ACM, 9–16. https://doi.org/10.1145/2024569.2024572

[9] Dean Michael Berris, Alistair Veitch, Nevin Heintze, Eric Anderson, and Ning Wang. 2017. XRay: A function call tracing system. 11th annual US LLVM Developers' Meeting.

[10] Derek Bruening, Timothy Garnett, and Saman P. Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2003)*. IEEE Computer Society, 265–275. https://doi.org/10.1109/CGO.2003.1191551

[11] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. IEEE Computer Society, 213–223. https://doi.org/10.1109/CGO.2011.5764689

[12] Justin Campbell and Mike Walker. 2020. GitHub - microsoft/onefuzz: A self-hosted Fuzzing-As-A-Service platform. https://github.com/microsoft/onefuzz. [Online; accessed 04-Nov-2021].

[13] Justin Campbell and Mike Walker. 2020. Microsoft announces new Project OneFuzz framework, an open source developer tool to find and fix bugs at scale - Microsoft Security Blog. https://www.microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/. [Online; accessed 04-Nov-2021].

[14] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. 2016. Living on the edge: rapid-toggling probes with cross-modification on x86. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 16–26. https://doi.org/10.1145/2908080.2908084

[15] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. 2017. Instruction punning: lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 320–332. https://doi.org/10.1145/3062341.3062344

[16] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (SP)*. 711–725. https://doi.org/10.1109/SP.2018.00046

[17] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. Ptrix: Efficient hardware-assisted fuzzing for cots binary. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 633–645.

[18] TIS Committee. 1993. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification. (1993).

[19] Mozilla Corporation. 2014. mozilla/sccache: sccache is ccache with cloud storage. https://github.com/mozilla/sccache. [Online; accessed 26-Feb-2022].

[20] Christian Dietrich, Valentin Rothberg, Ludwig Füracker, Andreas Ziegler, and Daniel Lohmann. 2017. cHash: Detection of Redundant Compilations via AST Hashing. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 527–538. https://www.usenix.org/conference/atc17/technical-sessions/presentation/dietrich

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

[22] Free Software Foundation. 2021. Labels as Values (Using the GNU Compiler Collection (GCC)). https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html. [Online; accessed 01-Nov-2021].

[23] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. 2015. Optimizing binary translation of dynamically generated code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 68–78. https://doi.org/10.1109/CGO.2015.7054188

[24] Marc Heuse. 2021. AFLplusplus/README.md at 3.14c · AFLplusplus/AFLplusplus · GitHub. https://github.com/AFLplusplus/AFLplusplus/blob/3.14c/README.md#cite. [Online; accessed 01-Nov-2021].

[25] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. 2018. INSTRIM: Lightweight instrumentation for coverage-guided fuzzing. In *25th Annual Network and Distributed System Security Symposium (NDSS)*.

[26] Apple Computer Inc. 1999. nlist.h - Apple Open Source. https://opensource.apple.com/source/xnu/xnu-1228.0.2/EXTERNAL_HEADERS/mach-o/nlist.h.auto.html. [Online; accessed 01-Nov-2021].

[27] Google Inc. 2021. dynamorio/drcovlib.c · DynamoRIO/dynamorio. https://github.com/DynamoRIO/dynamorio/blob/7595b777289b70a4752ecb6db5ca7987efeeaaaf/ext/drcovlib/drcovlib.c. [Online; accessed 14-Nov-2021].

[28] Microsoft Inc. 2021. PE Format - Win32 apps | Microsoft Docs. https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#export-address-table. [Online; accessed 01-Nov-2021].

[29] Yuseok Jeon, Wookhyun Han, Nathan Burow, and Mathias Payer. 2020. FuZZan: Efficient Sanitizer Metadata Design for Fuzzing. In *2020 USENIX Annual Technical Conference*. USENIX Association, 249–263. https://www.usenix.org/conference/atc20/presentation/jeon

[30] Yaron Keren. 2018. yrnkrn/zapcc: zapcc is a caching C++ compiler based on clang, designed to perform faster compilations. https://github.com/yrnkrn/zapcc. [Online; accessed 26-Feb-2022].

[31] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[32] Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. 2010. PEBIL: Efficient static binary instrumentation for Linux. In *IEEE International Symposium on Performance Analysis of Systems*

and Software. IEEE Computer Society, 175–183. https://doi.org/10.1109/ISPASS.2010.5452024

[33] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. 2018. PartiSan: Fast and Flexible Sanitization via Run-Time Partitioning. In *Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11050)*. Springer, 403–422. https://doi.org/10.1007/978-3-030-00470-5_19

[34] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *11th Joint Meeting on Foundations of Software Engineering*. 627–637. https://doi.org/10.1145/3106237.3106295

[35] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. 2022. PACSan: Enforcing Memory Safety Based on ARM PA. *CoRR* abs/2202.03950 (2022). arXiv:2202.03950 https://arxiv.org/abs/2202.03950

[36] Jie Liang, Yu Jiang, Mingzhe Wang, Xun Jiao, Yuanliang Chen, Houbing Song, and Kim-Kwang Raymond Choo. 2021. DeepFuzzer: Accelerated Deep Greybox Fuzzing. *IEEE Trans. Dependable Secur. Comput.* 18, 6 (2021), 2675–2688. https://doi.org/10.1109/TDSC.2019.2961339

[37] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 2022 IEEE Symposium on Security and Privacy (SP)(SP). IEEE Computer Society, Los Alamitos, CA, USA*. 154–170.

[38] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. ACM, 190–200. https://doi.org/10.1145/1065010.1065034

[39] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2013. System V Application Binary Interface. *AMD64 Architecture Processor Supplement, Draft v0* 99 (2013), 57.

[40] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: an open fuzzer benchmarking platform and service. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1393–1403. https://doi.org/10.1145/3468264.3473932

[41] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *IEEE Symposium on Security and Privacy (SP)*. 787–802. https://doi.org/10.1109/SP.2019.00069

[42] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*. ACM, 89–100. https://doi.org/10.1145/1250734.1250746

[43] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium (NDSS)*. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[44] Joel Rosdahl. 2010. Ccache — Compiler cache. https://ccache.dev/. [Online; accessed 26-Feb-2022].

[45] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*. 167–182.

[46] Kostya Serebryany. 2016. Sanitize, Fuzz, and Harden Your C++ Code. USENIX Association, San Francisco, CA.

[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference*. USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[48] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2011. Dynamic Race Detection with LLVM Compiler - Compile-Time Instrumentation for ThreadSanitizer. In *Runtime Verification - Second International Conference (Lecture Notes in Computer Science, Vol. 7186)*. Springer, 110–114. https://doi.org/10.1007/978-3-642-29860-8_9

[49] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *2005 USENIX Annual Technical Conference*. USENIX, 17–30. http://www.usenix.org/events/usenix05/tech/general/seward.html

[50] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy*. IEEE, 1275–1295. https://doi.org/10.1109/SP.2019.00010

[51] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 46–55. https://doi.org/10.1109/CGO.2015.7054186

[52] Todd L. Veldhuizen. 2003. *C++ Templates are Turing Complete*. Technical Report. Indiana University Computer Science.

[53] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. 2015. High System-Code Security with Low Overhead. In *2015 IEEE Symposium on Security and Privacy, SP 2015*. IEEE Computer Society, 866–879. https://doi.org/10.1109/SP.2015.58

[54] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. 2021. RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing. In *2021 USENIX Annual Technical Conference*. USENIX Association, 147–159. https://www.usenix.org/conference/atc21/presentation/wang-mingzhe

[55] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. 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*. IEEE, 328–337. https://doi.org/10.1109/ICSE-SEIP52600.2021.00042

[56] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *27th Annual Network and Distributed System Security Symposium (NDSS)*. https://www.ndss-symposium.org/ndss-paper/not-all-coverage-measurements-are-equal-fuzzing-by-coverage-accounting-for-input-prioritization/

[57] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. 2021. SanRazor: Reducing Redundant Sanitizer Checks in C/C++ Programs. In *15th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 479–494. https://www.usenix.org/conference/osdi21/presentation/zhang

[58] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. 2020. Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 858–870. https://doi.org/10.1145/3324884.3416572

[59] Zhiqiang Zuo, Kai Ji, Yifei Wang, Wei Tao, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. JPortal: precise and efficient control-flow tracing for JVM programs with Intel processor trace. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1080–1094.