Dredd: Scalable Source-Level Mutation Testing for Large C/C++ Codebases

Alastair F. Donaldson
Imperial College London
London, United Kingdom
alastair.donaldson@imperial.ac.uk

James Lee-Jones
Imperial College London
London, United Kingdom
james.lee-jones20@imperial.ac.uk

Jonathan Foo
Imperial College London
London, United Kingdom
jonathan.foo20@imperial.ac.uk

Abstract

We present Dredd, a tool for source-level mutation testing of large C/C++ codebases. Following the *mutant schemata* approach, Dredd applies a source-to-source transformation to a chosen set of files from the system under test (SUT), producing a *metamutant* that simulates a large number of distinct mutants, each selected via an environment variable. When a given mutant is selected, the metamutant behaves as if a specific edit (e.g. replacing a binary operator) had been applied. Because the metamutant is compiled and linked only once, mutation analysis across many mutants avoids expensive per-mutant recompilation and linking. We evaluate Dredd's scalability on the Clang compiler (a large C++ project) and the zstd compression library (a sizeable C project), and showcase an application of Dredd in compiler testing, using Dredd to compare the effectiveness of various fuzzers with respect to their ability to kill mutants.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Keywords: Mutation testing, mutant schemata, metamutant, Clang libTooling

ACM Reference Format:
Alastair F. Donaldson, James Lee-Jones, and Jonathan Foo. 2026. Dredd: Scalable Source-Level Mutation Testing for Large C/C++ Codebases. In *Companion Proceedings of the 2026 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '26), October 3–9, 2026, Oakland, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3837729.3840482

1 Introduction

Mutation analysis and mutation testing [2] have a long history as techniques for assessing and improving the adequacy of testing approaches [8] by determining whether a test input is capable of detecting synthetic changes to the source code of a system under test (SUT), such as removing a statement or changing an operator. However, the expense of applying mutation testing has limited its uptake: even with a small number of mutation operators, an industrial codebase can easily yield hundreds of thousands or millions of mutants. At this scale, the traditional idea of editing and recompiling the SUT source code separately for each mutant is infeasible if one wishes to assess the ability of a testing approach to *kill mutants* across the SUT by detecting the inserted changes.

We present Dredd, a mutation testing tool for C/C++ codebases aimed at practitioners who wish to assess and improve the quality of test suites and testing tools.[1] Dredd operates at the source level (rather than on bytecode or IR) and achieves the scalability required by large C/C++ codebases via the *mutant schemata* approach [23, 24]: instead of generating one program variant per mutant, Dredd applies a single source-to-source transformation to selected SUT files, rewriting them into a *metamutant* that encodes many distinct mutants simultaneously, guarded by dynamic checks. At runtime, the mutant of interest is selected via an environment variable holding the desired mutant ID. With no mutant selected, the metamutant behaves identically to the original SUT. Otherwise, it behaves as if the selected mutant had been applied. Because the metamutant is compiled and linked only once, mutation analysis over the entire mutant space avoids expensive per-mutant recompilation and linking.

We outline how to use Dredd (Section 2) and describe its design and implementation (Section 3). We then present two case studies (Section 4), applying Dredd to Clang/LLVM [15] (a large C++ project) and zstd [17] (a sizeable C project). We conclude with a discussion of related work (Section 5), future directions (Section 6), and tool availability (Section 7).

2 Using Dredd

Dredd is a command-line tool that takes the source files to be mutated as arguments. Because C/C++ source files only make sense in the context of a compilation environment (e.g. with header search paths provided via `-I` and preprocessor directives via `-D`), Dredd can also be pointed at a *compilation database* [13]—a JSON file specifying the compilation environment for each source file in a project. Compilation databases can be generated automatically by

[1] Dredd is named after the comic book character Judge Dredd, because Dredd assists in *judging* the adequacy of test suites and testing tools.

SPLASH Companion '26, Oakland, CA, USA

ACM ISBN 979-8-4007-2910-2/2026/10
https://doi.org/10.1145/3837729.3840482

build systems such as CMake [10], or via the build interception tool Bear [18]. As an example, the following command causes Dredd to mutate all `.cpp` files under the `Transforms` directory of the Clang/LLVM compiler, and relies (via the `-p` option) on a `compile_commands.json` compilation database being present under `build` (the build folder for the project):

```
dredd --mutation-info-file=info.json -p build \
  $(find llvm/lib/Transforms -name "*.cpp" | sort)
```

The `--mutation-info-file` option stores the ID of each mutant and the source code edit to which it is equivalent in a JSON file. Dredd provides a script for querying this information in a human-readable manner.

After mutation, the SUT should be recompiled using its standard build commands. In the case of Clang/LLVM, this leads to a compiled `clang` binary instrumented with mutants. To enable a particular mutant when executing the compiled binary, the user should set the `DREDD_ENABLED_MUTATION` environment variable. For example:

```
DREDD_ENABLED_MUTATION=42 clang example.c
```

causes mutant 42 embedded in the `clang` binary to be enabled when `clang` is used to compile `example.c`. If the environment variable is not set, no mutants are enabled and the SUT behaves as normal. However, the instrumentation that Dredd applies (outlined in Section 3) introduces runtime overhead even if no mutant is enabled.

Because a test input can only kill mutants it may execute, Dredd provides a *mutant tracking* mode that records (to a file) the IDs of mutants reached during execution, without enabling any mutants. A user can combine "tracking" and "mutated" builds of the SUT to allow for efficient mutation analysis: the former determines the subset of mutants reached by a given test, and the latter is used to assess whether the test kills each of these mutants in turn. Note that repeat runs may be needed to gain confidence in mutant tracking for systems that exhibit non-determinism or test flake.

Rather than coupling Dredd to a testing framework or computing metrics such as mutation score, we offer a simple CLI with utility scripts that users can integrate with their project's existing testing infrastructure.

3 Design and Implementation

We illustrate the design of Dredd using the following simple C++ method, from an `Account` class with a `balance` field:

```
void Deposit(int amount) {
  balance = balance + amount;
}
```

Traditionally, many distinct mutants can be derived from this example, such as the following:

```
void Deposit(int amount) {
  // deleted
}
```

```
void Deposit(int amount) {
  balance = balance % amount;
}
```

```
void Deposit(int amount) {
  balance <<= balance + amount;
}
```

```
void Deposit(int amount) {
  balance = 0;
}
```

These are examples of applying standard mutation operators (e.g. statement deletion, arithmetic operator replacement), leading to multiple SUT source code copies. Instead of producing a separate version of the SUT per mutant, Dredd implements a form of the *mutant schemata* idea [23, 24]: a source-to-source transformation that embeds *all* mutants into the SUT source code, allowing them to be toggled on and off. For the `Deposit` example, the Dredd-transformed version of the method is similar to the following:[2]

```
void Deposit(int amount) {
  if (!enabled(38)) {                    // Encodes  1 mutant
    replace_assign(28,                   // Encodes 10 mutants
      balance,
      replace_add(16,                    // Encodes 12 mutants
        replace_identifier(0, balance), // Encodes  8 mutants
        replace_identifier(8, amount)   // Encodes  8 mutants
      )
    );
  }
}
```

The **`enabled`** function takes a mutant ID and returns true if that mutant has been *enabled* via `DREDD_ENABLED_MUTATION` (Section 2). If mutant 38 is enabled, the assignment statement in the original `Deposit` is not executed, simulating statement deletion. The **`replace_assign`** function simulates replacing the `=` assignment operator in the original `Deposit` implementation with any of the 10 compatible assignment operators provided by C++ (e.g. `^=` and `>>=`). Value 28 is passed as a `base` argument, and **`replace_assign`** uses **`enabled`** to check whether `DREDD_ENABLED_MUTATION` lies in the range $[\texttt{base}, \texttt{base} + 10)$, behaving like one of the 10 replacement operators if so and the original `=` operator otherwise. The **`replace_add`** and **`replace_identifier`** calls work similarly, with **`replace_identifier`** simulating 8 mutants for each use of `balance` and `amount`, covering replacement by constants and application of unary operators (including side-effecting operators such as `++`, since the identifiers are l-values).

Supported mutation operators. Dredd supports simulation of statement deletion, binary and unary operator replacement, unary operator insertion, and replacement of expressions with constants. Expression-related mutations are restricted to the built-in numeric types of C/C++, and expressions that must be compile-time constants, such as array bounds and template arguments, cannot be mutated because Dredd's mutations are dynamically toggleable; this is a fundamental limitation of the mutant schemata approach.

[2] We have simplified this metamutant substantially (e.g. the functions Dredd emits have mangled names to avoid clashes with SUT names), but the example still conveys the intuition behind the design of Dredd.

Implementation details and optimisations. Dredd is implemented as a Clang tool using the libTooling framework [14], allowing it to be applied to a wide range of production C/C++ codebases. Significant engineering effort has gone into ensuring Dredd respects subtle C++ features, especially templates and template specialisation. Applying Dredd to Clang/LLVM, a large C++ project that makes extensive use of advanced language features, has helped achieve this aim. To reduce redundant mutants, Dredd leverages Clang's constant expression analysis (e.g. to avoid replacing an expression e with 0 when e is known to always evaluate to 0), and implements state-of-the-art techniques for avoiding mutation operator redundancy [9]. Testing the `DREDD_ENABLED_MUTATION` environment variable on every enabledness check would be inefficient. Instead, during a test execution the variable is read at most once per mutated source file, on the first check, and then cached in a file-local (static) variable. Replacing binary operators with function calls is non-trivial for the `&&` and `||` short-circuit operators, whose right-hand sides should not be evaluated by default. Dredd handles this in C++ via lambda expressions, and in C via a more involved scheme that we omit for space reasons.

4 Case Studies

In the following, all experiments were conducted on an Ubuntu Linux 24.04 KVM instance with 8 vCPUs and 64GB RAM on a host containing 2 × Intel Xeon E5-2690 v3 @ 2.60GHz CPUs. Timing-related results are averaged over 5 repeat runs.

4.1 Clang/LLVM (C++)

We report on our experience using Dredd to mutate all of the code in C++ source files under the `Transforms` directory in version 22.1.7 of the Clang/LLVM compiler. These files comprise 280,253 lines of source code (calculated using the `cloc` tool [1]), and capture the vast majority of optimisation logic in the compiler, including passes such as dead code elimination, function inlining and loop unrolling [16].

Mutation and recompilation overhead. Applying Dredd to these files takes 1855 s and yields 705,385 mutants. The full LLVM regression test suite passes when executed on the metamutant with no mutants enabled, giving confidence that it is semantically equivalent to the original.

A partial recompile of a release mode (optimisations enabled) build of Clang/LLVM after touching, but not mutating, all of these source files normally takes 428 s. This increases to 1238 s when the source files have been mutated using Dredd, a 3× recompilation overhead. This is a *one-off* cost: the mutant schemata approach means that, after one recompilation, any mutant can be dynamically enabled.

Runtime overhead. To assess the overhead introduced by Dredd's transformations, we measure the time taken to compile a non-mutated version of Clang 22.1.7 from scratch in release mode (excluding additional tools and tests) and 100 C programs randomly generated using Csmith [21, 25] (git commit 0cdc710) at the `-O3` optimisation level. Compiling in release mode and using `-O3` ensures that the code under `Transforms` is exercised. It takes 2743 s to compile Clang/LLVM from scratch with a non-mutated Clang/LLVM, rising to 3589 s when a mutated Clang/LLVM is used (1.3× overhead). It takes 53 s to compile the 100 generated Csmith programs (total) with a non-mutated Clang/LLVM, rising to 101 s when a mutated Clang/LLVM is used (1.9× overhead). These overheads are modest given the intrusive nature of Dredd's instrumentation, because mutating `Transforms` only affects optimisation, and not other compilation stages such as parsing, semantic analysis and code generation.

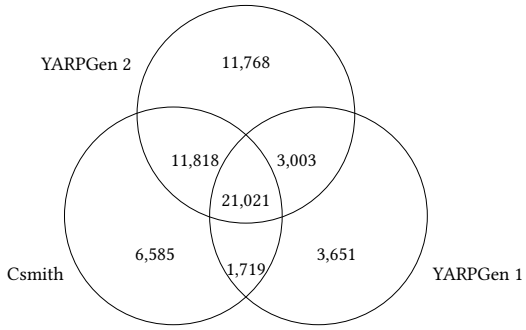

Figure 1. Mutants killed by Csmith and YARPGen v1/v2 during 1-week runs

Using Dredd to compare compiler fuzzers. We performed three 1-week experiments assessing the ability of Csmith, YARPGen v1 [7, 11] (git commit 30072f7) and YARPGen v2 [7, 12] (git commit e0f63b6) to kill mutants under `Transforms`. For each generator we launched 8 parallel workers that repeatedly: generated a C program; used a "tracking" build of Clang to determine the mutants reached when compiling at `-O3`; and iterated through each reached mutant m not yet killed, compiling the generated program with a "mutated" build of Clang at `-O3` and m enabled to determine whether it was killed. A mutant is considered killed if it causes compilation to crash, hang (assumed if compilation takes more than 5× as long as without the mutant), or produce object code that yields a different result from that produced by the non-mutated compiler. Figure 1 shows the number of mutants killed by each combination of tools after 1 week, revealing significant complementarity, with YARPGen v2 killing the most mutants. Figure 2 shows the cumulative number of mutants killed during the fuzzing runs, showing that the vast majority of mutants are killed during the first 24 hours, with the fuzzers making slow but steady progress thereafter. This application of Dredd suggests that mutation killing can provide an alternative to bug-finding ability when evaluating fuzzers, avoiding the problem that bugs found by earlier tools may no longer be available to distinguish newer ones.

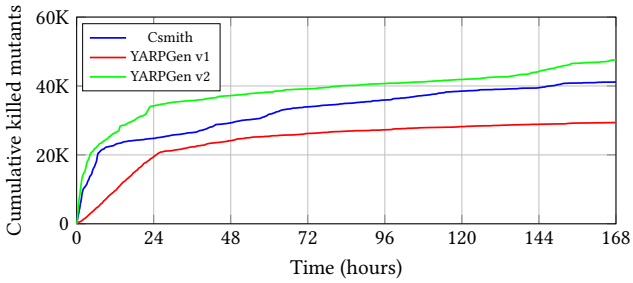

Figure 2. Cumulative count of Clang/LLVM mutants killed by compiler fuzzers during 1-week runs

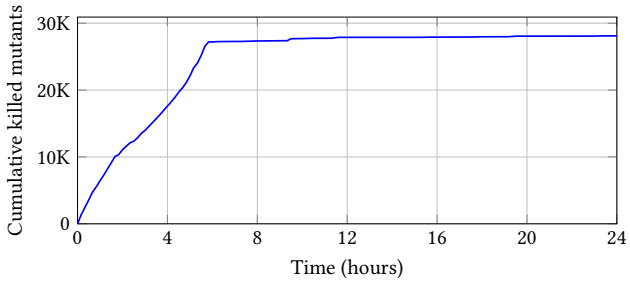

Figure 3. Cumulative count of zstd mutants killed by `datagen` during a 24-hour run

4.2 Zstandard (C)

We used Dredd to mutate all C code under the `compress`, `decompress` and `common` subdirectories of Zstandard (zstd), a widely used compression library from Meta [17]. These files comprise 16,591 lines of code (calculated using `cloc`).

Mutation and recompilation overhead. Applying Dredd to these files takes 5 s and yields 141,272 mutants. Confidence that the metamutant is semantically equivalent to the original is discussed under *runtime overhead* below.

A partial recompile in release mode after touching (but not mutating) the relevant source files normally takes 8 s. This increases to 437 s when the source files have been mutated using Dredd. This one-off 55× recompilation overhead is substantial, and principally arises during the compilation of `zstd_opt.c`, a large file that encodes 10,615 mutants.

Runtime overhead. We used both mutated and unmodified versions of zstd to compress and decompress 1000 data files randomly generated by the `datagen` tool that ships with zstd, confirming in each case that such a roundtrip leaves the data file intact (thus giving confidence that the metamutant is semantically equivalent to the original codebase). This took 29 s vs. 361 s without and with mutation, indicating a 12.5× overhead. Because zstd's sole responsibility is compression and decompression, and the corresponding core code was mutated, this high overhead is unsurprising. It could be reduced by sharding mutation of the SUT so that the union of shards incorporates all desired mutants while each shard contains only a subset, providing an interesting design point between the traditional "one program per mutant" and mutant schemata approaches.

Using compression roundtripping to kill mutants. We performed a 24-hour fuzzing run where 8 parallel workers repeated the process of generating a random data file (using `datagen`), determining the mutants reached by a compress-decompress roundtrip, and attempting to kill each such mutant in turn via the roundtrip. This led to 28,104 mutants being killed in total, but Figure 3 shows that the killing process largely saturated after 4.5 hours, with few mutants killed thereafter. Mutants reached but not killed via this process could be good targets for driving the construction of edge-case compression inputs to test zstd more thoroughly.

5 Related Work

Mutation testing is a large and mature field [8]. Although mutant schemata date to the early 1990s [23, 24], we believe Dredd is the first tool to realise the approach for realistic C/C++ codebases. MUSIC [20] targets C but follows the traditional "one program per mutant" approach, incurring per-mutant compilation and linking costs. A Google experience report [19] also follows a "one mutant, one program" model and scales by only mutating code touched by a change and left uncovered by its tests. Closest to Dredd is Mull [3], which operates on LLVM bitcode. While Mull originally JIT-compiled mutants separately, it has recently adopted a mutant schemata approach similar to Dredd's. The key difference is the mutation level: source-level mutation expresses mutants in a form that developers can understand, allows the use of any compiler, keeps the mutant space independent of compiler optimisations, and avoids build-system surgery. An advantage of Mull is that it supports any language targeting LLVM bitcode (including Rust). We have explored the synergy between mutation testing and fuzzing, including using mutation analysis to compare test-case generators, which has also been discussed in recent work [6].

6 Conclusions and Future Work

We have presented Dredd, a source-level mutation testing tool for C and C++ that uses mutant schemata to make mutation analysis practical for large codebases. Case studies on Clang/LLVM and zstd show that Dredd can mutate substantial systems with reasonable recompilation costs and manageable runtime overheads. Future work includes applying Dredd to a broader range of projects and investigating mutation-space sharding as a trade-off between per-mutant recompilation and instrumentation overhead.

7 Tool Availability

We provide a reproducibility artifact for this paper on Zenodo [5] and a demo video [4]. Dredd is open source (Apache 2.0) on GitHub [22], with usage instructions in the README.

Acknowledgements

We thank Amber Gorzynski for feedback on an earlier draft, and Google for support via gift funding.

References

[1] Al Danial. 2026. cloc: Count Lines of Code. Accessed: 2026-06-24. https://github.com/AlDanial/cloc

[2] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. doi:10.1109/C-M.1978.218136

[3] Alex Denisov and Stanislav Pankevich. 2018. Mull It Over: Mutation Testing Based on LLVM. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 25–31. doi:10.1109/ICSTW.2018.00024

[4] Alastair F. Donaldson. 2026. Dredd: Scalable Source-Level Mutation Testing for Large C/C++ Codebases (YouTube Video). Accessed: 2026-06-24. https://youtu.be/Yovf7ZJMDUc

[5] Alastair F. Donaldson, James Lee-Jones, and Jonathan Foo. 2026. Artifact for "Dredd: Scalable Source-Level Mutation Testing for Large C/C++ Codebases". doi:10.5281/zenodo.21640176

[6] Rahul Gopinath, Philipp Görz, and Alex Groce. 2022. Mutation Analysis: Answering the Fuzzing Challenge. *CoRR* abs/2201.11303 (2022). arXiv:2201.11303 https://arxiv.org/abs/2201.11303

[7] Intel Corporation. 2026. YARPGen: Yet Another Random Program Generator. Accessed: 2026-06-24. https://github.com/intel/yarpgen

[8] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678. doi:10.1109/TSE.2010.62

[9] René Just and Franz Schweiggert. 2015. Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators. *Softw. Test. Verification Reliab.* 25, 5-7 (2015), 490–507. doi:10.1002/STVR.1561

[10] Kitware, Inc. 2026. CMake. Accessed: 2026-06-25. https://cmake.org/

[11] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 196:1–196:25. doi:10.1145/3428264

[12] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1826–1847. doi:10.1145/3591295

[13] LLVM Organization. 2026. JSON Compilation Database Format Specification. Accessed: 2026-06-25. https://clang.llvm.org/docs/JSONCompilationDatabase.html

[14] LLVM Organization. 2026. LibTooling. Accessed: 2026-06-25. https://clang.llvm.org/docs/LibTooling.html

[15] LLVM Organization. 2026. The LLVM Compiler Infrastructure. Accessed: 2026-06-25. https://github.com/llvm/llvm-project

[16] LLVM Project. 2026. LLVM Passes Documentation: Transform Passes. Accessed: 2026-06-24. https://llvm.org/docs/Passes.html#transform-passes

[17] Meta. 2026. Zstandard. Accessed: 2026-06-25. https://github.com/facebook/zstd

[18] László Nagy. 2026. Bear. Accessed: 2026-06-25. https://github.com/rizsotto/Bear

[19] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and René Just. 2022. Practical Mutation Testing at Scale: A view from Google. *IEEE Trans. Software Eng.* 48, 10 (2022), 3900–3912. doi:10.1109/TSE.2021.3107634

[20] Duy Loc Phan, Yunho Kim, and Moonzoo Kim. 2018. MUSIC: Mutation Analysis Tool with High Configurability and Extensibility. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 40–46. doi:10.1109/ICSTW.2018.00026

[21] The Csmith Project Authors. 2026. Csmith. Accessed: 2026-06-24. https://github.com/csmith-project/csmith

[22] The Dredd Project Authors. 2026. Dredd. Accessed: 2026-06-24. https://github.com/mc-imperial/dredd

[23] Roland H. Untch. 1992. Mutation-based software testing using program schemata. In *Proceedings of the 30th Annual Southeast Regional Conference, 1992, Raleigh, North Carolina, USA, April 8-10, 1992*, Mladen A. Vouk, Douglas S. Reeves, and Cherri M. Pancake (Eds.). ACM, 285–291. doi:10.1145/503720.503749

[24] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. 1993. Mutation Analysis Using Mutant Schemata. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis, ISSTA 1993, Cambridge, MA, USA, June 28-30, 1993*, Thomas J. Ostrand and Elaine J. Weyuker (Eds.). ACM, 139–148. doi:10.1145/154183.154265

[25] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. doi:10.1145/1993498.1993532

Received 2026-06-26; accepted 2026-07-25