SN4KE: Practical Mutation Testing at Binary Level

Mohsen Ahmadi
Arizona State University
pwnslinger@asu.edu

Pantea Kiaei
Worcester Polytechnic Institute
pkiaei@wpi.edu

Navid Emamdoost
University of Minnesota
navid@cs.umn.edu

***Abstract*—Mutation analysis is an effective technique to evaluate a test suite adequacy in terms of revealing unforeseen bugs in software. Traditional source- or IR-level mutation analysis is not applicable to the software only available in binary format. This paper proposes a practical binary mutation analysis via binary rewriting, along with a rich set of mutation operators to represent more realistic bugs. We implemented our approach using two state-of-the-art binary rewriting tools and evaluated its effectiveness and scalability by applying them to SPEC CPU benchmarks. Our analysis revealed that the richer mutation operators contribute to generating more diverse mutants, which, compared to previous works leads to a higher mutation score for the test harness. We also conclude that the reassembleable disassembly rewriting yields better scalability in comparison to lifting to an intermediate representation and performing a full translation.**

I. Introduction

In many contexts software is accompanied with test suites when delivered to the customer. In such cases, the test suite is the primary and mostly the only leverage to assure correctness of the software. This demonstrates the importance of test suite adequacy. Evaluating such adequacy requires a separate analysis which is known as mutation analysis. Such analysis generates *mutants* of the original program via making small changes either at source or IR level. Such mutants then are ran through the test suite. If the test suite successfully differentiates a mutant from the original program based on its observed output, it is said that it has *killed* that mutant. The ratio of killed mutants to all tried mutants is called *mutation score* which is the metric to describe the adequacy of the test suite.

In order to create mutants, many researches proposed various mutation operators which are the rules for changing statements or instructions in the original program to get a new mutant. A good mutation operator is representative of a real world bug that may be introduced as a result of errors either in development or the building process. For example, consider a bug-fixing patch which is accompanied with a test. A mutation operator to evaluate the test's adequacy would be undoing the effect of the patch, e.g if the patch introduces a bounds check, the mutation operator causes skipping the check.

In many scenarios the program source code is not available like in proprietary software or shared libraries. In these cases the mutation tool should still be able to generate mutants from the original binary. Binary level mutation has been explored previously [16], [4]. The major challenge on binary mutation tools is practical and scalable binary rewriting which in turn imposes limitation on the binary mutation operators. In order to generate mutants from the original binary, the mutation tool needs to restore a higher level code representation like assembly and then apply the mutation operator which may change the instructions address layout. Binary rewriting has found many applications in program analysis, security and etc. Mutation analysis is another application of binary rewriting which requires generation working mutants by making changes at instruction level.

Binary rewriting techniques have been extensively incorporated in security enforcement tools like [25], [37], [38], [15] where the instructions are aligned, inserted or replaced to enforce measurements like control flow integrity or fault isolation and etc. In recent years reliable binary rewriting techniques (like Ddisasm [17] or Ramblr [33]) have been proposed which expand the domain of such measurements applications to the programs available only in binary format. Previously researchers have demonstrated the applicability of reassembleable disassembly for binary mutation [16]. They used Uroboros [34] and were able to mutate binaries from SPEC CPU. Limitations imposed by Uroboros obstructs the practicality of such application, more specifically authors reported fragility of the tool in a way that it worked for specific compilation options. Their implemented mutation operators are limited to only conditional jump and move instructions.

In this paper, we revisit the notion of binary mutation in light of recent advancements in binary rewriting to implement a practical binary mutation tool that can support real-world binaries. Additionally, we employ a richer set of mutation operators that span over conditional, logical, and arithmetic instructions in order to have a more rigorous evaluation of the test harness. Our binary mutation tool named SN4KE is accessible at https://github.com/pwnslinger/sn4ke/.

The rest of the paper is as follows: in section II we discuss the related work on the subject of binary mutation analysis of tests. section III presents our approach and design of SN4KE, specifically the set of mutation operators and the binary rewriting engine. section IV describes our evaluation of SN4KE on SPEC CPU benchmarks and the comparison of performance of the two binary rewriting tools we used. section V discusses some of interesting challenges we had in adapting rev.ng for our purpose. Finally, section VII concludes the paper.

Workshop on Binary Analysis Research (BAR) 2021
21 February 2021, Virtual
ISBN 1-891562-69-X
https://dx.doi.org/10.14722/bar.2021.23017
www.ndss-symposium.org

TABLE I: Mutation Operator Classes

<table>
<tr><th>Mutation Class</th><th>Description</th></tr>
<tr><td>Arithmetic</td><td>Replace arithmetic assignment operators from the set of {+=, -+, *=, /=}
Replace with an operator from the set of {+, -, *, /, %}</td></tr>
<tr><td>Logical</td><td>Substitute with another bit-wise logical operator from {ˆ, |, &}
replace with a logical assignment from {ˆ=, |=, &=}
Substitute the connector with another logical operator from {&&, ||}</td></tr>
<tr><td>Conditional</td><td>Substitute any conditional jump with an unconditional branch to force taking the branch
taking the fall-through (instruction following the branch) edge of branch by NOPing the condition</td></tr>
<tr><td>Constants</td><td>Replace any immediate value c with one another constant from set {-1, 0, 1, -c, c+1, c-1}</td></tr>
<tr><td>Skip</td><td>Replace instructions from set Arithmetic, Logical, and Conditional with a NOP operator to skip the execution of that operator</td></tr>
</table>

II. Related Work

Generating variants of a program has been employed in the context of security: fuzzing tools like T-Fuzz [28] negate the conditions to pass the blocking conditions and explore new paths; additionally, tools like EvilCoder [29] LAVA [14] insert exploitable bugs in the program to provide testing corpus for vulnerability detection tools. In a broader context, mutation analysis tends to generate variations of the software and use it as a data set to evaluate the adequacy of tests to catch such variants. Authors in [16] employed Uroboros [34], a reassembleable disassembly binary rewriting tool to recover the assembly representation of a binary and apply the mutation at the assembly level. They applied a limited set of mutation operators mainly focusing on flag-use instructions and used the generated mutants to evaluate the adequacy of tests for SPEC CPU benchmarks. Their findings demonstrates that the tests designed for performance benchmarking catch fewer mutants compared to tests designed for code coverage.

As we are focused on mutation analysis at binary level, therefore we describe works related to mutation analysis and binary rewriting in the following subsections.

A. Mutation Analysis

Since its introduction [8], [10], mutation testing has been studied extensively. Numerous mutation tools have also been proposed for different programming languages like C [20], Java [24], C# [12], Fortran [36], SQL [31]. Mutation on intermediate representation (IR) has been studied as well [11]. In [19] authors compared source-level and IR-level mutation testing and observed closely correlated mutation scores.

Mutation testing relies on two fundamental hypotheses: Competent Programmer Hypothesis [1], and Coupling Effect hypothesis [26]. The former hypothesis states that the programmer tends to develop the program with minimal faults and close to the correct. Therefore, a mutation analysis with simple mutation operators still can simulate actual faults that are introduced by programmer. The latter hypothesis states that complex program faults can be de-coupled into simpler ones. Therefore a test that catches simple faults can catch a high percentage of complex ones, where complex faults are associated to making more than one change into the original program. The mutation operators that we used in this paper are based on the ones in [18] which target four main categories: conditional instructions, logical instruction, arithmetic instructions, and constant values. We explain these mutation operators in more details in subsection III-A.

Offutt et al [27] formulated the conditions that a test suite can kill a mutant. First of all, the mutated instruction should be *reachable* by at least one test input in the test suite. Second, the reached mutated instruction should enter the execution to an incorrect state. Third, the incorrect state should propagate and reach the program output.

Depending on the mutation operators and level of mutation application, the number of generated mutants varies. Hariri et al [19] reported that source-level mutation produced fewer mutants compared to IR-level mutation. Our experience is in alignment with this finding where binary level mutation may create numerous mutants. Not all the generated mutants have same quality in evaluating test adequacy. In this paper we filter out the *trivial* mutants which are the ones that fail to execute on any input data. Such mutants has no contribution in evaluating the effectiveness of the test suite.

B. Binary Rewriting

In this section, we will go through the publicly available binary rewriting solutions and compare their approach with regard to structural recovery, data type extraction, and limitations on supported architectures. Binary rewriting is the process of modifying a compiled program in such a way that it remains executable and functional without accessing the source-code. Binary Mutation is the process of purposefully introducing faults into a program without having the source-code. While in the definition of that there is not any criteria on soundness of the resulting binary, for the purpose of test suite evaluation we require the injected fault passes some trivial tests.

There are two methods of modifying the binary, one statically and the other over the span of a program execution which is called dynamic. In static approach, we keep a copy of modifications on a separate file on disk. However, dynamic rewriting which is denoted as instrumentation applies the modifications at the runtime.

Dynamic rewriters like PIN [23] and DynamoRio [6] apply instrumentation at defined locations in memory. Compared to static rewriter, at runtime there is only one code block to translate and deal with. However, depending on the implementation, frequent control flow changes or context switches between rewriter built-in virtual machine (VM) and operating system adds up overhead. Another kind of dynamic rewriting

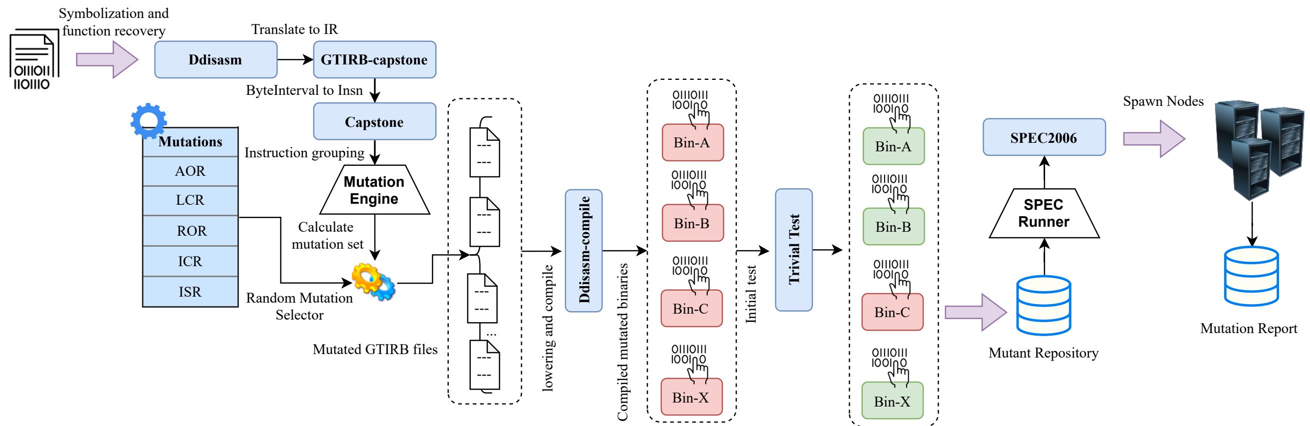


Fig. 1: Workflow of SN4KE: First, we pass the binary under test to Ddisasm for reconstructing the relocation table by identifying the instruction boundaries using backward and forward traversal techniques described in their paper [8]. As a result, we can retrieve the lifted binary in GTIRB representation which contains symbolized addresses for references and variables. Gtrib-capstone is the built-in rewriter, and we modified the code to fix some bugs related to shifting the symbolic expression of blocks following the insertion point. Gtirb represents binary as a module contains sections, and each section is a byte-interval. Blocks point to ranges into the byte interval to represent their bytes that either is DataBlock or *CodeBlock* objects. Hence, a byte-interval encapsulates all of the blocks for that section. Next, the mutation engine finds the matching operators and calculates all possible mutations that fall under mutation class categories. Then by applying each mutation resulting Gtirb file passed to Ddisasm-compiler to re-compile the disassembly. The resulting binaries should pass initial tests before passing them out to the SPEC Runner.

has been introduced by dynamic translating of program to an IR [5]. Tools like Valgrind or QEMU can lift the binary to an IR and perform code mutations on top of the generated IR. Then, they can translate it back to apply those modifications during execution. While this approach seems to be effective, it suffers from runtime overhead.

Our focus is on static rewriting schemes and hence we dedicate the rest of this section to it. There are three known static rewriting schemes. The oldest one is based on *detouring* at assembly level. *Detouring* works by hooking out the underlying instruction. There are two flavors of detouring technique, *patch-based instrumentation* and *replica-based instrumentation* [35]. *Patch-based instrumentation* replaces the instruction with an unconditional branch into a new section containing instrumentation, replaced instruction, and a control flow transfer back to the patch point. Detouring is a direct rewriting and is ISA dependent which makes the approach inconvenient. This approach introduces a high performance degradation given the two control transfers at patch points.

Replica-based instrumentation method places jump instructions at control flow changing destinations to a replicated code section containing both a copy of the original code and instrumentation. All memory references in this section are modified to maintain less control flow transfers between original and replicated section. While the performance of this approach is better compared to the *patch-based*, the size of the resulting binary is noticeably increased. [2] incorporated replica-based instrumentation in the malware domain by hooking APIs often used by malware authors to detect analysis environment.

Reassembleable disassembly is another static rewriting technique which works by recovering relocatable assembly code. Hence, instrumentation could be inlined and reassembled back to a working binary. This approach first introduced by UROBOROS [34] and then expanded and improved by Ramblr [33]. This approach enhances the performance since inlined assembly avoids inserting control flow changing instructions at instrumentation points. As a result, performance penalty caused by jump instructions is alleviated. Ddisasm [17] is the state-of-the-art tool for reassembling disassembly developed in Datalog and combines novel heuristics in function recovery and data access pattern.

Among the three options, we chose Ddisasm as our candidate for reconstructing disassembly because of the following reasons: 1) Compared to Ramblr, Ddisasm is about five times faster. 2) UROBOROS scans the data section linearly and considers any machine word-sized buffer whose integer representation falling in a memory region as a memory reference. This assumption under the compiler optimization introduces False-positive and False-negatives [33]. 3) Ramblr improved the *content classification* by applying strong heuristics like *localized value-set analysis* and *Intra-function data dependence analysis*. To achieve a higher accuracy in Control Flow Graph (CFG) recovery Ramblr heavily relies on using symbolic execution which slows down the rewriting process. Apart from Ramblr heuristics, ddisasm incorporated register value analysis (RVA) as an alternative over traditional Value Set Analysis (VSA). In addition, they introduced Data Access Pattern (DAP) analysis which uses the results of def-use analysis combined with the results of register value analysis for a refined register value inference at any given data access point.

Full-translation approach works by translating a low-level machine code to a high-level intermediate representation (IR) using a compiler-based front-end for architecture independent binary rewriting. The process of translating the binary to the IR is called *lifting*, while assembling the IR back to a working

executable is denoted as *lowering*. Advantages of lifting the binary to a high-level IR are two folds. First, relying on IR makes the rewriting framework ISA-agnostic, as a result leading to support more architectures. Second, providing the ability to apply program analysis techniques like Value Set Analysis (VSA) [3] and optimization passes like Simple Expression Tracker (SET) and Offset Shifted Register Analysis (OSRA) conveniently [13]. On the other hand, complete translation suffers from changing the structural integrity such as cache locality and CFG.

rev.ng [13] relies on full binary translation by lifting the binary to TCG (the IR used in QEMU) and translating TCG to LLVM IR to benefit from more advanced transformation and analysis passes for CFG and function boundary recovery. Frameworks like angr [32] use lifting to apply more advanced binary analysis on top of the intermediate-level representation. While rev.ng has shown to be useful for modifying the binary file [21], tools like angr do not have the functionality of lowering the resulting transformations back to the binary. Moreover, rev.ng heavily relies on code pointers for identifying function entry points and leverages value-set analysis for a more precise value boundary tracking.

III. APPROACH

Mutation analysis has been widely studied to evaluate the effectiveness and quality of test suites concerning code coverage and semantic integrity of program. Mutation testing works by replacing operators or operands with a list of candidates inheriting similar behavior. These studies apply mutations directly on source-code or by lifting the code to an intermediate representation (IR) [19]. In the previous research [16], authors applied Uroboros [34], a static binary rewriter for the purpose of mutation generation. Uroboros recovers program structures from stripped binaries and provide an API for recovering the Control Flow Graph (CFG) and call graphs of a binary. Since Uroboros is capable of recovering the lost relocation information, it is possible to inline assembly in the middle of the binary. While Uroboros provides a relocatable disassembly, we discussed in Section II-B that it is not scalable to real-world examples and lacks correctness and soundness in symbolization process.

In this paper, we introduce SN4KE, a light-weight scalable binary mutation framework backed with a rich set of mutation operators analogous to source-level mutation engines like SRCIROR [19]. Relying on binary rewriters which does not work on top of an IR or does not provide any API to work with the intermediate representation, decreases the portability of mutation engine over architectures. SN4KE is a conglomerate of two powerful binary rewriting frameworks, rev.ng [13] and Ddisasm [17]. Ddisasm works on top of the Grammatech Intermediate Representation for Binaries (GTIRB) [30], which lifts the binary to an IR and maps it to an assembly representation. Hence, this mapping facilitates the re-assembling of binary. To incorporate the power of LLVM IR passes for applying more powerful mutations, we integrated rev.ng with SRCIROR. rev.ng lifts the binary to Tiny Code Generator (TCG), QEMU's IR, and para-lifts the result to LLVM IR. In Section IV, we evaluate the benefit and challenges of applying these two engines in program mutation domain. In the following, we describe the mutation operators used over the course of our experiments.

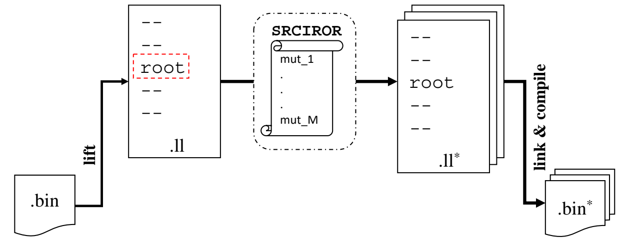


Fig. 2: Integration of rev.ng and SRCIROR to generate mutant binaries

A. Mutation Operators

Numerous mutation generation engines introduced on source-code and to provide the same level of confidence in binary mutation, having an engine close to those is needed. Emamdoost et. al. [16] manifested a mutation engine based on Uroboros which only covers conditional branch mutations. To make the assumption close and realistic to errors might occur during development and software patches, they created mutations for either fall-through or taken branch. While control flow modifiers constitute a large amount of effective instructions, we see that there are lots of faults might happen because of Off-by-One (OBO) or integer wrap-around errors which are related to arithmetic operations.

Hence, In our mutant generation, we consider five classes of mutation operators: replacing one arithmetic operator with another different arithmetic operator (AOR), replacing a logic operator with another different logic operator (LCR), replacing the constant operand in an operation with another different constant value (ICR), and replacing the predicate of a comparison with another different predicate (ROR), and skipping instruction by replacing with NOP operator (ISR). We apply only first order mutations, i.e., for each mutated binary only one mutation operator is applied.

As Table I shows, in arithmetic replacements, we substitute every occurrence of an arithmetic operations with an operator from the set of {+, /, *, %}. For logical connectors, we divide them into three sub-categories of logical assignments, logical operators, and bitwise operators of set {and, or, xor, not}. For constant replacement we look at the operands passed to an operator and check the immediate values. Then we select a random constant *c* and generate mutants for any combination of {c+1, c-1, 0, 1, -1, -c}. For conditional operators, on the assembly level, we either take the next instruction coming after the branch by NOPing the instruction or changing the predicate to take the branch using an unconditional branch.

To apply the above mentioned mutations, we transfer the target binary to another domain with different abstraction. As discussed in the following sections, we use two different binary rewriters using which we can apply the mutations either at the level of assembly code or at the level of LLVM IR. Even though our mutation operators are similar in the two tools, the difference in the level of application of mutations leads into different results. More specifically, the mutations applied by Ddisasm have a one-to-one relationship with the mutations in

the binary whereas the mutations applied by rev.ng at the level of LLVM IR go through the compiling process and therefore may result in modifications to multiple instructions in the binary. Given the optimizations performed on the IR level for rev.ng size and structure of the resulting binaries are different compared to the original binary.

B. Rewriting Strategies

1) Ddisasm: Ddisasm is the main module used for recovering the lost structures and symbolization of unresolved memory addresses. For compatibility with other binary analysis tools, Ddisasm incorporates Google's Protobuf protocol to serialize recovered data and analysis results in GITRB. One distinction between GTIRB and other IRs like VEX, LLVM IR [22], and BAP's BIL [7] is that mainly IRs used to represent the semantics of assembly instruction with more verbosity level to include architecture modifications to registers and CPU flags. While, the main idea behind GTIRB is to represent the binary structure from a high-level while preserving the assembly content. Simply, GTIRB acts like a container of binary analysis in conjunction with assembly content.

We show our framework pipeline in Figure 1. In the beginning, framework takes a binary as input and passes it over to Ddisasm to generate a relocatable disassembly in GTIRB representation. GTIRB contains different abstraction for storing information of *Module*, *Symbols*, *SymbolicExpressions*, *Sections*, *ByteIntervals*. As part of our contribution, we modified the default rewriter in Gtirb-capstone to fix the symbolic expressions for the Code and Data blocks following the insertion points and any reference to shifted symbols in the Auxiliary Data.

Capstone is an ultimate disassembly engine designed for binary analysis and security research. Unfortunately, for the purpose of our research Capstone had a limited set of instruction grouping which was not sufficient for mutation analysis. We added new set of groups to better cover our analysis. At the moment of writing the paper, Capstone only supported JUMP, CALL, RET, and INTERRUPT groups. We added Arithmatic, Logical, and Bitwise grouping. Next, decoded instructions from *ByteInterval* passed to the Mutation Engine to calculate all the possible first-level mutations based on the instruction category.

For the purpose of our experiments, we randomly sampled 1000 mutations and passed the resulting mutated GTIRB file to Ddisasm-compiler. During this process, some of the mutations failed the compilation step due to either missing *.eh_frame* section or conflicting *.ctor* and *.dtor* sections. In the latter case, further investigation proved that default linker put elements from *.ctor* and *.dtor* into *.init_array* and *.fini_array* sections lead to misalignment in sections. Finally, to make sure the mutated binary will not crash because of Segmentation Fault or corrupted heap bins, we pass the mutated binaries to a trivial test. This test basically executes the binary with inputs like *-h* or *-v* to assert the healthy state.

2) rev.ng: To apply our mutations at the level of LLVM IR, we chose rev.ng. rev.ng's ability to lift the binary to LLVM IR provides a few advantages. First, manipulating LLVM IR is simple and comes with a vast set of open-source tools. Second, applying changes to the IR language of a compiler will not allow generating code that is not compilable hence filters out the unsuitable mutations.

In Figure 2, we show our methodology. We first lift the binary using rev.ng's lifter. We then apply our mutations from the aforementioned set of mutation operators to the lifted binary. This is implemented in the form of LLVM's compiler optimization passes. We used the SRCIROR's IR mutation setup [19] for this purpose. Once we have the mutated LLVM IR of the binary, we run LLVM's linker and compiler to generate the mutated binary.

From the rev.ng setup, the `translate` tool is the main rewriter. `translate` is able to translate a binary from one target architecture to another. It works by lifting the binary to LLVM IR using its own lifter and compiling for another architecture. As a result of lifting, rev.ng adds extra functions (QEMU helper functions) to the lifted LLVM IR while keeping the code of the original binary inside a function called `root`. To integrate SRCIROR and rev.ng, we adjusted the SRCIROR setup for LLVM version 10.0.1 to be consistent with Clang's version used in rev.ng tools. We also modified it to mutate only the parts from the original binary file, i.e. only the root function. We then separated the lifting from the link and compile processes in `translate`. After lifting, we use SRCIROR to enumerate all the possible points and types of mutation with our set of mutation operators. Next, we apply each of the possible mutations one at a time, generating mutated IRs. Finally, we run the linker and compiler processes in `translate` to generate the mutant binaries.

IV. Evaluation

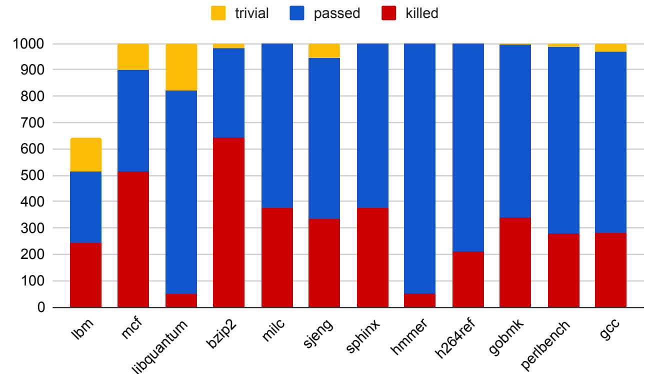


Fig. 3: Mutation results for SPEC 2006 benchmark with test input set

To demonstrate the SN4KE's scalability and practicality, we used it to generate binary mutants of SPEC CPU 2006 benchmarks compiled for x64 architecture. SPEC CPU is a collection of programs along with three different data sets to evaluate performance of compiler, processor, and or memory. The provided input sets are designed for performance evaluation rather than fault detection or code coverage. That said, the benchmarks are carefully observed to match the expected output. Three different input sets mainly differ in size and purposes: the `test` input is used for checking the benchmark's functionality, `train` input is used at build time

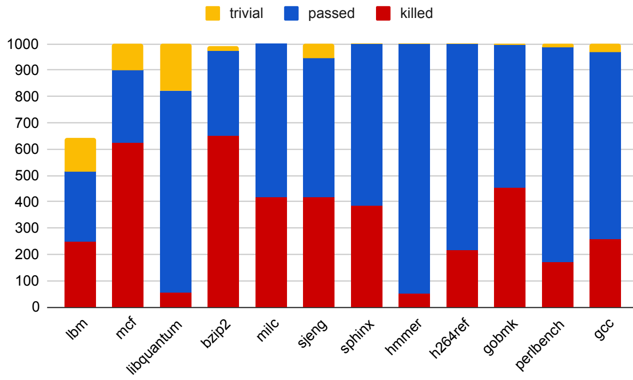


Fig. 4: Mutation results for SPEC 2006 benchmark with train input set

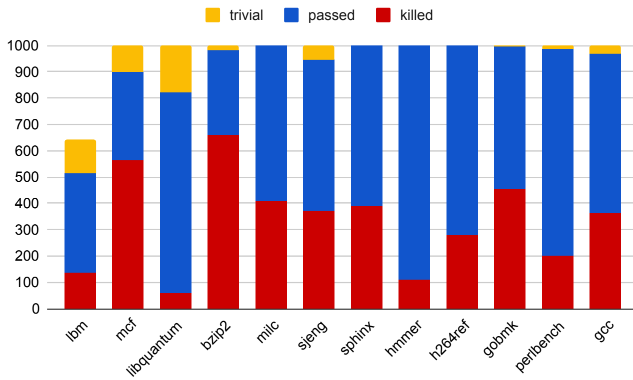


Fig. 5: Mutation results for SPEC 2006 benchmark with ref input set

of the benchmark for feedback directed optimizations, and `ref` input is the largest and actual input data used for performance evaluation.

Based on the approaches described in section III we generate as many as possible mutants, but for the purpose of mutation analysis, we randomly select a subset of 1000 mutants and calculate mutation score per each input data set. Figure 3 shows the ddisasm mutation engine results on the `test` input set. The experimental results on the `train` set is presented in Figure 4 and Figure 5 includes the results of `ref` set. The proportion of killed mutants for the `test` input set is relatively lower than for the `train` and `ref` input sets. That is mainly because the `test` input is smaller in size and more simple in a way that does not cover most portions of the code. `perbench` is an outlier here, as the `test` input set has kill more mutants compared to other two input sets. That is because based on SPEC CPU documentations, the `test` input set is derived from the actual test harness shipped with Perl 5.8.7.

The number of mutants generated for each benchmark directly correlates with the code size. However, the mutants' diversity in terms of applied mutation operator follows a common pattern among all the benchmarks. Compared to the previous work by Emamdoost et al., we were able to generate more mutants due to our richer mutation operator set. Except for `lbm`, we could generate more mutants than the sampling rate threshold. While `lbm` code base only consists of 1500 instructions, we can generate 641 mutants, which means 42% of instructions fall under at least one mutation category.

TABLE II: Mutation score of `test`, `train`, and `ref` sets. Mutations score denoted as the number of killed mutants over the total number of mutants. This measure used in assessing the quality of test suite.

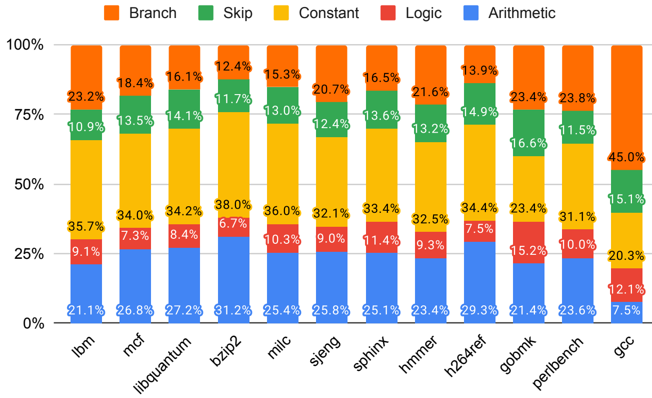
Binaries	**Mutation score**		
	test	**train**	**ref**
lbm	37.8%	38.8%	21.8%
mcf	51.4%	60.0%	56.4%
libquantum	5.2%	5.7%	6.0%
bzip2	64.4%	65.7%	66.1%
milc	37.7%	41.6%	40.9%
sjeng	33.3%	42.0%	37.0%
sphinx	37.7%	38.4%	38.8%
hmmer	5.1%	5.0%	10.8%
h264ref	20.9%	21.6%	28.0%
gobmk	34.0%	45.5%	45.5%
perlbench	27.8%	16.9%	20.4%
gcc	28.2%	25.5%	36.4%

We classify mutants in three categories: passed, killed, and trivial based on the initial health-check tests and expected functionality test results. Passed mutants are the ones that generate outputs consistent with original benchmark binaries. Killed mutants are defined as those that fail to generate the same output as the baseline binary because of either calculation errors or falling into an infinite loop. The set of mutants which fail the initial health-check tests or never make the execution due to a segmentation fault or pipe error are classified as trivial. Initial health-check test is a short light-weight input for each binary. For example, in case of `gcc` is an empty file.

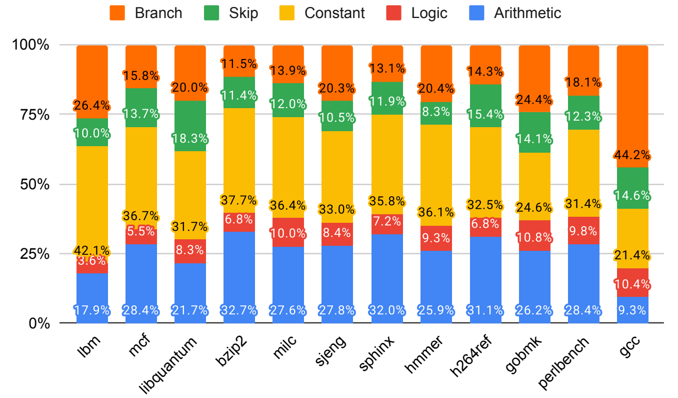
Infinite loops happen when we apply a control-flow related mutation or changing the logical operator following a comparison. Unbounded loops put the program in a non-returning state. Hence, to cover these edge cases, we applied a timeout chosen twice of the original runtime.

Figure 6a provides the mutant's breakdown based on the operator used in each mutant generated by `ddisasm` rewriter. The three most common classes of mutants are arithmetic switching, constant switching, and instruction skip (NOP). This observation constitutes to our initial thought on the diversity of new mutants beyond control flow changing instructions. Dominance of branch mutations in `gcc` comes from the supremacy of control-flow instructions compared to other binaries which comprises the 22% of the whole instructions.

Breakdown of killed mutants on the `ref` test at Figure 6b follows a constant pattern complies with Figure 6a. Table II manifest the mutation score which is the percentage of mutants killed over the total number of mutants. Compared to [16] results, generally we observed higher mutation scores. For example, in `gcc` they reported a score of less than 15% for a large code base while ours is 36.4%. Based on their results, mutation score and binary size has inverse relation. The greater the size of the binary is, the lower mutation score they reported. Since they only relied on conditional branches for mutation,



(a) Selected set mutation breakdown.



(b) Killed mutants of `ref` set breakdown.

Fig. 6: Mutation Breakdown of 1000 randomly selected mutants over five mutation categories resulting by `ddisasm` engine. In all of the cases, Arithmetic, NOP (Skip), and Constant operations have greater share compared to others. Greater the size of the binary is the more mutants we are going to expect.

TABLE III: Size and number of possible mutations on the lifted LLVM-IR by Revng

benchmark	IR size	possible mutations
lbm	14MB	53215
mcf	14MB	47390
libquantum	24MB	177310
bzip2	30MB	270217
milc	46MB	476663
sjeng	53MB	496435
sphinx	63MB	699322
hmmer	103MB	1228840
h264ref	177MB	2110590
perlbench	367MB	4759791
gcc	986MB	13117690
gobmk	283MB	3507198

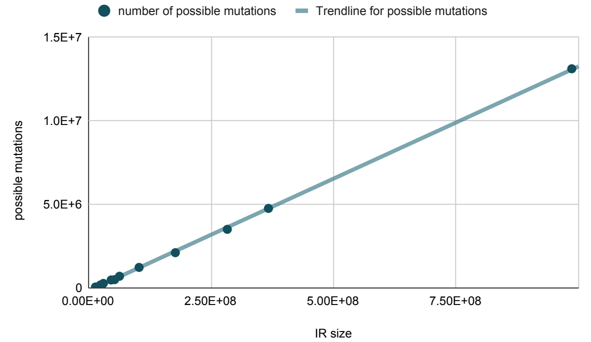


Fig. 7: Number of possible mutations by SRCIROR vs. size of the lifted IR by rev.ng

their score for smaller binaries are larger which shows the effectiveness of branch mutation in relatively small binaries.

Based on [16], they reported problems with compilation flags of `gcc` and `gobmk` binaries and their rewriter was limited to a specific version of GCC compiler. In both of our approach, because of powerful binary structure recovery provided by both `Rev.ng` and `ddisasm`, we never had such an issue. This shows the portability of our approach compared to their in stable binary mutation of real-world binaries.

V. Discussion

Mutant generation using the rev.ng tool had scalability issues that prohibited us from thoroughly analyzing the mutants generated from SPEC CPU 2006.

Mutant binary size increase becomes a serious issue, specifically when the original binary has a moderate code size. Table IV shows the size of the rewritten binaries in SPEC 2006 test suite using rev.ng and Ddisasm. The rewritten binaries from rev.ng had a size of from 10× to 70× the original binary, while the Ddisasm rewriter kept the rewritten binaries stable in size. We tracked down the reason behind size expulsion and found that during the lifting process to the TCG IR, rev.ng imports a large portion of helper function calls in the resulting binary. rev.ng authors suggested applying the function isolation pass to reduce the binary size by dropping the root function. CRISPR [9] is a tool that allows static binary patching using rev.ng with minimized binary size overhead. While we didn't test either of these approaches, they believe even after applying this method, we still should expect a binary greater than the original one.

Mutant generation time was another factor that rendered rev.ng a less appealing rewriting tool for binary mutation. As mentioned before, the number of possible mutants goes beyond millions for binaries with moderate code size like `gcc`. In such cases the generation time per mutant becomes a decisive factor on practicality and scalability of the binary mutation tool. Generation of the IR from binaries by rev.ng took less than a minute for small binaries such as `lbm` and `mcf` but about 20 minutes for `perlbench` and 30 minutes for lifting `gcc`. Furthermore, the generated IR itself is of significant size as pointed out in Table III. The bulky size of the lifted IR made the total number of applicable mutations very large. As

TABLE IV: Overhead in code size resulted from Revng and Ddisasm binary rewriters for SPEC 2006 binaries

benchmark	original size	Revng		Ddisasm	
		rewritten size	rewrite overhead	rewritten size	rewrite overhead
lbm	22kB	1.2MB	54.5×	22kB	1×
mcf	23kB	1.2MB	52.2×	22kB	0.96×
libquantum	51kB	3.5MB	68.6×	46kB	0.9×
bzip2	69kB	4,1MB	59.4×	68kB	0.99×
milc	142kB	5.7MB	40.1×	134kB	0.94×
sjeng	154kB	8.1MB	52.6×	149kB	0.97×
sphinx	198kB	7.5MB	37.9×	196kB	0.99×
hmmer	314kB	13MB	41.4×	308kB	0.98×
h264ref	566kB	22MB	38.9×	552kB	0.98×
perlbench	1.2MB	47MB	39.2×	1.5MB	1.25×
gcc	3.6MB	127MB	35.3×	3.5MB	0.97×
gobmk	3.9MB	39MB	10.0×	4.3MB	1.1×

Figure 7 shows, number of possible mutations by SRCIROR is linearly related to the size of its IR input with the trendline of $0.0134x - 172510$ and coefficient of determination of $R^2 = 1$.

VI. Future Work

In this research, we addressed the limitations of binary mutation by employing more robust binary rewriting approaches and also, adopting a richer set of mutation operators inspired by source-level mutation. However, we did not explore the possible vulnerabilities that could be introduced by each of those mutations that resemble a real-bug in a program. Such observations can be helpful for example to further investigate which mutation works better for proof-testing patches in a binary. For example to generate mutants resembling double fetch vulnerabilities by performing a light-weight def-use chain analysis on the operators working on same memory locations. One possible mistake causing this issue is the incorrect replacement of lines of code during a patch. While a manual test-case might fall short to test the patch comprehensively, a good binary mutation engine can test for these corner cases.

VII. Conclusion

In this paper, we proposed SN4KE, a new binary mutation tool. SN4KE's modular design allowed us to adapt and compare the two latest binary rewriting tools: Ddisasm and rev.ng. Based on our implementation and evaluation, Ddisasm proved to be more practical and scalable in terms of both resulting mutant binary size and mutant generation time. SN4KE is able to mutate any binary independent of the original compilation configuration in use. Compared to the previous works on binary mutation, SN4KE enjoys a richer set of mutation operators. These operators are better representative of real world faults, and thanks to the scalable underlying binary rewriting technique, are easily applicable to any binary program paving the way to generate many mutants. Applying SN4KE to SPEC CPU 2006 benchmarks, we demonstrated its applicability and effectiveness in terms of generating diverse set of mutants. Such diverse mutants evaluated the adequacy of tests for SPEC CPU binaries.

References

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation analysis." Georgia Institute of Technology School of Information and Computer Science, Tech. Rep., 1979.

[2] M. Ahmadi, K. Leach, R. Dougherty, S. Forrest, and W. Weimer, "Mimosa: Reducing malware analysis overhead with coverings," 2021.

[3] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *International conference on compiler construction*. Springer, 2004, pp. 5–23.

[4] M. Becker, C. Kuznik, M. M. Joy, T. Xie, and W. Mueller, "Binary mutation testing through dynamic translation," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, June 2012, pp. 1–12.

[5] ——, "Binary mutation testing through dynamic translation," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 2012, pp. 1–12.

[6] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012, pp. 133–144.

[7] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.

[8] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "The design of a prototype mutation system for program testing," in *Proceedings of the AFIPS National Computer Conference*, vol. 74, 1978, pp. 623–627.

[9] F. Cremonese, "CRISPR: Binary Editing with High Level Languages," Nov. 2020. [Online]. Available: https://github.com/revng/crispr

[10] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[11] A. Denisov and S. Pankevich, "Mull it over: Mutation testing based on llvm," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2018, pp. 25–31.

[12] A. Derezińska, "Advanced mutation operators applicable in c# programs," in *Software Engineering Techniques: Design for Quality*, K. Sacha, Ed. Boston, MA: Springer US, 2007, pp. 283–288.

[13] A. Di Federico, M. Payer, and G. Agosta, "rev.ng: a unified binary analysis framework to recover CFGs and function boundaries," in *Proceedings of the 26th International Conference on Compiler Construction*, 2017, pp. 131–141.

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

[15] N. Emamdoost and S. McCamant, "The effect of instruction padding on SFI overhead," in *Binary Analysis Research (BAR)*. Internet Society, 2018.

[16] N. Emamdoost, V. Sharma, T. Byun, and S. McCamant, "Binary mutation analysis of tests using reassembleable disassembly," in *Binary Analysis Research (BAR)*. Internet Society, 2019.

[17] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[18] F. Hariri and A. Shi, "Srciror: A toolset for mutation testing of c source code and llvm intermediate representation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 860–863.

[19] F. Hariri, A. Shi, V. Fernando, S. Mahmood, and D. Marinov, "Comparing mutation testing at the levels of source code and compiler

intermediate representation," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 114–124.

[20] Y. Jia and M. Harman, "Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language," in *Testing: Academic Industrial Conference - Practice and Research Techniques (taic part 2008)*, Aug 2008, pp. 94–98.

[21] P. Kiaei, C.-B. Breunesse, M. Ahmadi, P. Schaumont, and J. van Woudenberg, "Rewrite to reinforce: Rewriting the binary to apply countermeasures against fault injection," *arXiv preprint arXiv:2011.14067*, 2020.

[22] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[24] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "Mujava: A mutation system for java," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 827–830. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134425

[25] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture." in *15th USENIX Security Symposium*. USENIX Association, 2006.

[26] A. Offutt, "The coupling effect: Fact or fiction," in *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, ser. TAV3. New York, NY, USA: ACM, 1989, pp. 131–140. [Online]. Available: http://doi.acm.org/10.1145/75308.75324

[27] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, pp. 165–192, 9 1997. [Online]. Available: https://doi.org/10.1002/(SICI)1099-1689(199709)7:3⟨165::AID-STVR143⟩3.0.CO;2-U

[28] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[29] J. Pewny and T. Holz, "Evilcoder: automated bug insertion," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 214–225.

[30] E. Schulte, J. Dorn, A. Flores-Montoya, A. Ballman, and T. Johnson, "Gtirb: intermediate representation for binaries," *arXiv preprint arXiv:1907.02859*, 2019.

[31] J. Tuya, M. J. Suarez-Cabal, and C. de la Riva, "Sqlmutation: A tool to generate mutants of sql database queries," in *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, Nov 2006, pp. 1–1.

[32] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 8–9.

[33] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again." in *NDSS*, 2017.

[34] S. Wang, P. Wang, and D. Wu, "Uroboros: Instrumenting stripped binaries with static reassembling," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 236–247.

[35] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, "From hack to elaborate technique—a survey on binary rewriting," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–37, 2019.

[36] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *Journal of Systems and Software*, vol. 31, no. 3, pp. 185–196, 1995.

[37] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 79–93.

[38] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352.