The realities of software development practice have made it increasingly important to have tools that can deal with software in binary (executable) format. One major driver of this trend is security: besides analyzing commercial or adversarial code, even with one’s own code, important security properties like exploitability depend on details of compilation and hardening and can only be accurately assessed on the final binary. But also outside of security, and even in high-assurance and embedded systems domains, modern developers rely extensively on third-party binary-only libraries and other components. This reliance makes it important for developers to be able to carry out as many software engineering tasks as possible even without source code. In this paper we consider a class of tasks related to software testing, and demonstrate that by harnessing recently-developed techniques for flexible binary rewriting, we can perform them effectively given just a binary.

Testing is one of the most commonly used approaches to improve and maintain software quality, but one challenge is that it can be difficult to measure the quality of software tests. The main goal of testing is to find bugs, but in the common steady state where our test suite passes on the current version of software, how can we assess tests’ bug-finding abilities? A test suite can be designed to be able to catch the bugs that have arisen in software in the past, but more important while also harder to tell is how it will perform in finding so-far-unknown bugs. One approach is structural coverage, for instance requiring that tests cause each instruction in a program to be executed. Coverage is necessary for bug-finding but not sufficient, as the requirements to trigger a bug may include a combination of control-flow decisions as well as properties of data. Automating a process closer to bugs themselves is the intuition for a technique called mutation analysis.

A mutation operator defines a way of making a small change to a piece of software, often inspired by the changes that differentiate a small bug from the corresponding corrected program. The result of applying mutation operators to various locations in a program is a set of mutants, programs which may have slightly different behavior from the original. Mutation analysis of a test suite assesses how many of the mutants can be differentiated from the original program by their behavior on one or more test inputs; a test that distinguishes a mutant is also said to kill it. The fraction of mutants killed is called a mutation score, with a higher score indicating a more complete test suite. Mutation analysis is a stronger, semantic criterion compared to structural coverage, and it can be used to evaluate a single test suite, a test generation approach, or a coverage criterion.

Most mutation analysis systems are designed for source code, or for intermediate-level representations like Java bytecode or LLVM bitcode. Instead we build a binary mutation tool that operates on ELF x86 binary programs, so that mutation analysis can be performed without needing source code for the tested software. Binary mutation is an instance of binary rewriting, which can be challenging because finished binaries are not designed to support further modification; they have already had internal references resolved into numeric addresses and offsets, so it is hard to move existing instructions, making room for new ones, while preserving the program’s behavior. We deal with this challenge by applying a recently-proposed approach to binary rewriting called reassembleable disassembly, in which the challenging aspects of binary rewriting are addressed in a reusable tool that converts a binary back into an assembly language format with symbolic references. This
assembly format, similar to what was produced by the original compiler, is still at the instruction level, but it is straightforward to modify because a standard assembler can be used to re-resolve the references and labels back to new but consistent numeric addresses.

Reassembleable disassembly is appealing because it provides a clean separation of concerns: only the disassembler needs to perform whole-program analysis such as recovering control-flow graphs. The separate assembly-language rewriter can be simple; it can be initially prototyped and tested on compiler-generated assembly, and it can be written in any language without interfacing with a specialized API. It is easy to examine and work with the effects of the rewriting in the form of a textual diff. Specifically we have implemented our binary mutation tool in conjunction with the open-source reassembleable disassembler Uroboros; taking advantage of the effort that has gone into Uroboros, we found that only modest additional implementation was required to build a mutation tool. Our current system implements mutation operators that force conditional instructions (conditional jumps, moves, and sets on x86) to be either taken or not taken unconditionally in the mutant; the reassembleable disassembly architecture would make adding other binary mutation operators straightforward as well. We made an open-source release of our prototype tool at https://github.com/navidem/binarymutation.

We evaluate our mutation analysis tool by applying it to two sets of experiment binaries and corresponding test suites. As examples of general-purpose software and to show scalability, we apply our tool to the SPEC CPU 2006 C benchmarks, evaluating the supplied benchmark inputs as a test suite. These inputs were designed primarily to test performance instead of correctness, but our results illustrate some expected general trends, such as that larger programs are more challenging to test completely. As a more specific application, we also apply our mutation tool to examples of safety-critical embedded control software with test suites generated from data-flow models, which have previously been used in research on binary-level test coverage criteria. With these examples, we can also compare the mutation scores assessed by our system with those measured by source-level mutation.

A. Motivating Example

An example illustrates how binary mutation provides an assessment of the quality of the test suite for a binary. The GNU Compiler Collection GCC [40] is a set of compilers commonly used for C and C++ code. The gcc binary (technically cc1) is also one of the benchmark programs from SPECInt2006 used in our evaluation. GCC gives special treatment to several common library functions, but older versions had a limitation in which GCC’s partial internal information for three functions (bcmp, bzero, and fputs) interfered with enforcing function prototypes for those functions present in the code. This was fixed in GCC’s development branch in February 2002.¹ The fix [41] for this bug was made by introducing a branch with new functionality for the case when the new condition was true. A regression test case was also introduced as part of this fix to detect the presence of this bug. Figures 1 and 2 show the change made at source and binary levels. Figure 3 shows the test input which is a C program. When this C program is compiled by the buggy GCC binary, no errors will be reported for the incorrect number of arguments given to bcmp, bzero, and fputs. The buggy version of GCC can also be simulated by mutating a fixed version to disable the branch that was added in the bug fix. Specifically binary mutation can achieve this by converting the jne instruction on line 5 of Figure 2 to an unconditional jump. If a test input like the one shown in Figure 3 is not present in the test suite, the mutant simulating this bug will not report a compilation error for other C/C++ code that misuse bcmp, bzero, or fputs. Such a GCC mutant will therefore not be killed. Such a mutant, when not killed by the test suite, is an indication of a limitation of a test suite. (In fact, both GCC’s internal regression tests prior to the discovery of this bug and the SPEC benchmark inputs show this limitation.) This example shows how binary mutation can help users of binary components assess the quality of their test suite by constructing mutants that execute control-flow mutated from the original binary. Our reassembleable disassembly-based binary mutation tool produces such mutants by using a mutation operator that produces two mutants for every conditional branch instruction: one that converts the conditional jump into a unconditional jump, and another that converts the unconditional jump into a no-op instruction.

An example illustrates how binary mutation provides an assessment of the quality of the test suite for a binary. The GNU Compiler Collection GCC [40] is a set of compilers commonly used for C and C++ code. The gcc binary (technically cc1) is also one of the benchmark programs from SPECInt2006 used in our evaluation. GCC gives special treatment to several common library functions, but older versions had a limitation in which GCC’s partial internal information for three functions (bcmp, bzero, and fputs) interfered with enforcing function prototypes for those functions present in the code. This was fixed in GCC’s development branch in February 2002.¹ The fix [41] for this bug was made by introducing a branch with new functionality for the case when the new condition was true. A regression test case was also introduced as part of this fix to detect the presence of this bug. Figures 1 and 2 show the change made at source and binary levels. Figure 3 shows the test input which is a C program. When this C program is compiled by the buggy GCC binary, no errors will be reported for the incorrect number of arguments given to bcmp, bzero, and fputs. The buggy version of GCC can also be simulated by mutating a fixed version to disable the branch that was added in the bug fix. Specifically binary mutation can achieve this by converting the jne instruction on line 5 of Figure 2 to an unconditional jump. If a test input like the one shown in Figure 3 is not present in the test suite, the mutant simulating this bug will not report a compilation error for other C/C++ code that misuse bcmp, bzero, or fputs. Such a GCC mutant will therefore not be killed. Such a mutant, when not killed by the test suite, is an indication of a limitation of a test suite. (In fact, both GCC’s internal regression tests prior to the discovery of this bug and the SPEC benchmark inputs show this limitation.) This example shows how binary mutation can help users of binary components assess the quality of their test suite by constructing mutants that execute control-flow mutated from the original binary. Our reassembleable disassembly-based binary mutation tool produces such mutants by using a mutation operator that produces two mutants for every conditional branch instruction: one that converts the conditional jump into a unconditional jump, and another that converts the unconditional jump into a no-op instruction.

**Fig. 1:** Fixing a bug in gcc’s duplicate_decls method to use argument type information from declared prototypes of bcmp, bzero, and fputs

```c
1️⃣ static int
duplicate_decls {
2️⃣    newdecl, olddecl,
3️⃣    different_binding_level)
4️⃣    tree newdecl, olddecl;
5️⃣    int different_binding_level;
6️⃣    ...
7️⃣    /* begin patch */
8️⃣    else if (TYPE_ARG_TYPES (oldtype) == NULL
9️⃣        && TYPE_ARG_TYPES (newtype) != NULL) {
10️⃣        ...
11️⃣    } /* end patch */
```

**Fig. 2:** Binary changes introduced by the patch shown in Figure 1

```c
1️⃣ 805c9c5: mov 0x18(%esp),%eax
2️⃣ 805c9c9: mov 0xc(%eax),%ecx
3️⃣ 805c9cc: test %ecx,%ecx
4️⃣     ; TYPE_ARG_TYPES (oldtype) == NULL
5️⃣ 805c9ce: jne 805bda4 <duplicate_decls+0x134>
6️⃣ 805c9d4: mov 0xc(%edi),%eax
7️⃣ 805c9d7: test %eax,%eax
8️⃣     ; TYPE_ARG_TYPES (newtype) != NULL
9es⃣ 805c9d9: je 805bda4 <duplicate_decls+0x134>
```
II. RELATED WORK

Program mutation is used to generate variations of a program. Techniques like LAVA [18] or EvilCoder [38] tend to insert either synthetic or actual bugs in the program source code in order to generate vast corpora of vulnerable programs. Such corpora serve as data set to evaluate and compare the quality of vulnerability detection techniques. The inserted bugs must be exploitable meaning that there should be an input that triggers it. To this end, vulnerability insertion tools employ different program analysis techniques to find a path from the input to the location of the inserted bug. T-Fuzz [37] incorporates program mutation to negate sanity checks that prevent a fuzzer from exploring deep paths. This way the fuzzer can find hidden bugs in the program.

Since binary mutation using reassemblable disassembly is related to binary rewriting as well as mutation testing, we describe work related in these two different areas of research in the below two subsections.

A. Mutation Testing

Mutation Testing [10], [23] is a fault-based testing technique with the primary objective of finding a “mutation score”, given a set of mutation operators. The technique relies on generating a set of mutants that mirror mistakes commonly made by programmers, an assumption that has been found to be empirically reliable [5]. Mutation testing has been applied to many programming languages such as Fortran [50], C [22], Java [29], C# [17], SQL [45], and AspectJ [14], as well as the LLVM IR [16]. The mutation operators used in this paper are similar to the “Remove Conditionals” mutation operator used by PIT [13], a popular mutation testing tool for Java programs. Mutation testing relies on two fundamental hypotheses [23]. First, the Competent Programmer Hypothesis [4] states that programmers tend to be competent, in the sense that they tend to write programs that are close to being correct. This hypothesis allows simulating faults in the original program by means of creating simple mutations which can be close to actual faults that would be introduced by programmers. Second the Coupling Effect hypothesis [15] states that if a test suite is capable of distinguishing programs with simple faults, it is also capable of distinguishing programs with more complex faults. This hypothesis has been offered by Offutt et al. [34] who found that complex faults are coupled to simple faults. A test set that detects all simple faults will also detect a high percentage of complex faults, where complex faults are ones corresponding to making more than one change to the original program. Our binary mutation tool introduces a class of simple faults by replacing conditional instructions with both unconditional variants. While we base our binary mutation testing tool on these two hypotheses, we plan in the future to investigate which binary mutation operators can represent simple mistakes made by competent programmers.

There are three fundamental properties that a test suite should have to kill a binary mutant, as set out by Offutt et al [36]. (1) A mutated instruction should be reachable by at least one test case in the test suite. (2) A mutated instruction that is reached should cause the mutant binary to have an incorrect state. This is called the necessity property. (3) The incorrect state introduced by the mutated instruction should propagate to the set of outputs observed by the mutation testing tool for the mutant binary. This is called the sufficiency property. In our evaluation, we found that a number of live mutants were not reachable. Future work could automate investigation of the necessity and sufficiency properties of the test suite for binary mutation.

It is possible to have a large number of mutants for each subject program, but not every mutant contributes equally to test effectiveness. This observation makes it desirable to reduce the number of mutants to be tested by either removing mutants that are equivalent to one another or the subject program, or by removing mutants that otherwise don’t contribute to test effectiveness. The exclusion of mutation operators that create a large number of mutants without reducing the mutation score, as by Offutt et al. [35] and Namin et al. [31], [32], [33] is a direction along which we plan to extend binary mutation in the future. It is also desirable to not generate or exclude mutants that are either equivalent to one another or equivalent to the original subject program. However, this semantic criterion is known to be an undecidable problem in general [9]. Several practical approaches to solve this problem of duplicate and equivalent mutants include reducing it to the feasible path problem [36], using program slicing to detect equivalent mutants [21], and using trivial compiler equivalence [24]. We plan to use these techniques to extend our binary mutation tool to detect duplicate and equivalent mutants.

B. Binary Rewriting

Binary mutation is an example of binary rewriting, which can be performed either statically (i.e., producing a modified executable file) or dynamically (producing new code as the program executes). Tools like Pin [27] for x86 binaries and DynamoRIO [8] for x86 and ARM allow the control flow [42], [39] to be changed at runtime. Dynamic binary translation tools that use an intermediate representation (IR), such as QEMU and Valgrind, can also be modified to implement mutations, though the flexibility of an IR is less important for simple mutations, and uses of an IR tends to make the translated code less efficient. It is also possible to change the binary statically using static binary rewriting tools such as DynInst [43].

Binary mutation based on dynamic binary translation has been demonstrated by Becker et al. [7], who used QEMU. Dynamic binary translation avoids some of the analysis challenges of static rewriting because it translates a block of code
only when that code is about to be executed, and it can still index instructions by their addresses in the original binary; however it imposes a runtime overhead on all code execution. By comparison static rewriting like what we preform imposes a one-time translation cost, but has little to no runtime overhead, which is advantageous if the mutated binary will be used for a long test suite. Our mutation operators are a subset of the mutation operators implemented by Becker et al. Becker et al. also added improvements to reduce mutant execution by killing mutants on the first observed deviation from the original program or with an adaptive timeout. We could reuse these ideas in the future to optimize our mutant executions.

In the reassembleable disassembly approach, a reusable tool extracts relocatable assembly from the binary without the use of debugging or relocation information. Uroboros [47], [48] was the first end-to-end system to provide such functionality. To make a disassembly reassembleable (relocatable), the main challenge is determining whether an immediate is used as an integer or as a label referencing another location. The process of identifying references in the disassembly is called symbolization. Uroboros performs its symbolization by identifying four types of symbol references: code to code, code to data, data to code, and data to data. Assuming that the original binary would not make invalid memory accesses, Uroboros filters out any immediate which falls out of address space allocated for the binary. In addition, to be a valid reference into the code section, an immediate must be the start of an instruction which is recovered by raw disassembly. Using these two conditions, Uroboros is able to symbolize correctly any immediate in the code section.

Similar but typically more challenging is data section symbolization, where Uroboros makes 3 assumptions: 1) Any reference stored in the data section depending on the machine architecture must be 4-byte or 8-byte aligned. 2) Binary rewriting does not need to change the layout of the data section, therefore no need to symbolize data to data references. 3) Any data to code reference is either a function pointer or jump table entry. Assumption 1 is always applied while assumptions 2 and 3 are configurable in the Uroboros implementation.

Uroboros also recovers parts of control flow structure via direct transfers. The authors of Uroboros demonstrated the usability of Uroboros in binary instrumentation by providing a trace profiling and diversification use cases.

Using the aforementioned assumption, Uroboros is able to symbolize a binary with a very low false positive and false negative. But still, there can be cases in which Uroboros fails to produce a correct binary and manual adjustments were needed. Motivated by examples of corner cases where Uroboros fails, Ramblr [46] (built on top of angr [44]) implements additional analyses to improve symbolization performance, aiming to avoid false positives and false negatives and to cover a more diverse set of binaries. Ramblr tries to decrease the number of candidate immediates for symbolization via local data flow and value set analysis. It tries to identify data in the code and perform type recognition on immediates. For example, if Ramblr can identify a 4-byte data block as a float, it should not be symbolized as a pointer. It also uses an array size recovery to identify more complex data structures and avoid symbolizing such locations. This way Ramblr is able to cover more corner cases for example introduced by compiler optimization or value collisions.

Another approach to making disassembly reliable is the superset disassembly approach of Multiverse [6]. In this technique, avoiding any assumption about the usage of memory addresses or immediates, disassembly is performed at every byte offset of the text section. The linear disassembly from each byte offset ends whenever it reaches an already visited byte offset or an invalid instruction. This way a superset of the actual disassembly is obtained. To make this superset reassembleable, a mapping from an address in original binary into the new binary is maintained. This mapping is dynamically looked-up at runtime to resolve any indirect control transfers in the new binary. A cost for the reliability provided by Multiverse is a larger code size expansion compared to other static rewriting approaches.

### III. Approach

![Fig. 4: An overview of our reassembleable disassembly-based binary mutation tool](image)

We present an overview of our reassembleable disassembly-based binary mutation tool in Figure 4. Given a binary-only representation of a subject program, a binary mutation tool can choose to either rewrite the binary or lift the binary semantics to an intermediate but higher level, mutate it, and recreate the binary from intermediate level representation. We chose the latter approach but only lift the binary to assembly code since it is still semantically close to the original binary code, suitable for mutation that changes a few instructions and might insert some, but leaves a majority of instructions unchanged. Our approach takes two inputs: (1) a subject program in binary form, (2) a test suite that has a set of test inputs and its corresponding set of expected outputs. Our binary mutation tool first disassembles the input binary. It then performs mutation on the input binary using a fixed set of mutation operators. Next, it reassembles each mutated assembly program to get a mutated binary. Finally, for every set of test inputs in the test suite, it runs every mutated binary
with the test input and checks if the mutated binary’s output matches the output expected for that test input. If at least one test input causes the mutated binary’s output to not match the expected output, then that mutant is considered killed. Once this process is complete for all combinations of test inputs and mutants, a mutation score is computed by dividing the number of mutants killed at the end of this process by the total number of mutants.

### TABLE I: Mutation operators to exclude a source-level branch

<table>
<thead>
<tr>
<th>Instruction(s)</th>
<th>Mutation rules used to substitute the original instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jcc</td>
<td>unconditional jump to Jcc target address</td>
</tr>
<tr>
<td></td>
<td>no-op to let execution continue to fallthrough</td>
</tr>
<tr>
<td>SETcc</td>
<td>move one-byte 1 to the destination</td>
</tr>
<tr>
<td></td>
<td>move one-byte 0 to the destination</td>
</tr>
<tr>
<td>CMPVcc</td>
<td>unconditionally move the source to the destination</td>
</tr>
<tr>
<td></td>
<td>no-op to skip the move</td>
</tr>
<tr>
<td>adc</td>
<td>perform only the addition operation in the original instruction</td>
</tr>
<tr>
<td></td>
<td>perform the original addition and increment the result</td>
</tr>
<tr>
<td>sbb</td>
<td>perform only the subtraction in the original instruction</td>
</tr>
<tr>
<td></td>
<td>perform the original subtraction and decrement the result</td>
</tr>
</tbody>
</table>

Our binary mutation tool performs mutation for three categories of x86 instructions and two other instructions, `adc` and `sbb`. Table I shows the mutation operators performed by our binary mutation tool. These mutation operators are applied to flag-use instructions as defined by Byun et al. [11]. We use the intuition that a binary mutation should simulate a source-level bug when possible. One category of source-level mistakes commonly made by programmers can be represented by reverting branches introduced as part of a patch. By creating mutant binary code that does not execute bug-fixes inside such branches, the mutant binary simulates the bug. This intuition was illustrated by the branch introduced in the bug-fix presented in Figure 1. We do not attempt to guess which direction of a conditional instruction is more likely to represent a bug fix: instead our tool generates two mutants for each conditional instruction, one as if the condition were always true, the other as if it were always false.

### IV. IMPLEMENTATION

Many source-level mutation operators have been proposed, and eventually a general-purpose binary mutation tool should support analogues of as many of them as feasible. Though mutations are generally always local changes to a binary, they might often require replacing instructions with longer ones, especially because on x86 architectures the length of an instruction can depend on minor details such as the size of an immediate value or the distance to a branch target. (For instance, consider replacing `x <= 255` with `x <= 256`.) Among the mutation operators of Table I, `adc` and `sbb` cause such artifact. For the case of `adc`, the second mutation rule requires replacing the instruction with an `add` followed by a `inc` instruction. The second mutation rule for `sbb` requires replacing the instruction with a `sub` followed by a `dec` instruction. When an instruction gets longer, it becomes challenging to modify the binary in place; replacing old code with a jump to new code can be complicated if the jump itself is longer than the old code, or if a location within the old code might have been a branch target. Dynamic binary instrumentation may be used as in [7], but it has some of its own disadvantages including runtime overhead. With the recent efforts to generate reassembleable disassembly, changes in a binary can be made elegantly and efficiently. We first lift the binary code into its assembly representation and then apply a mutation operation, and at the final step, the mutated assembly is reassembled into a binary.

We implemented our mutation analysis tool on top of Uroboros². (We discuss a few other tools that we considered in Section VI.) We found Uroboros to perform acceptably on small and mid-sized binaries. After extracting reassembleable disassembly, we applied the mutation operators described in Section III to generate mutants. For each mutant, the only functional difference from the original binary is in the mutated instruction. Once we got the mutants, we apply them to the test suites and measure the interactions between tests and mutants.

In order to measure the test suite quality, we collect two main metrics. The first metric is the mutation score or the rate of killed mutants. One detail of our mutation score definition should be mentioned. A mutation operator may introduce some fundamental flaw in a way that the mutant fails no matter what input is provided. Such mutants are called `trivial` and have no value in measuring the quality of the test suite. We separate such trivial mutants before applying the test suite, and only non-trivial mutants are used in the mutation score.

The second metric we consider is mutant reachability, meaning whether the test input reaches the mutated instruction or not. It is evident that for killed mutants this reachability is satisfied; otherwise, the mutant’s behavior would be the same as the original binary. But for live mutants (those that are not killed) we want to know how many of them are actually reached by the test input.

To measure the mutation score we may compare the mutant’s output versus the expected output of the original binary. In order to measure the live mutants reachability, we need to record the dynamically executed instructions for a specific input to check if the mutated instruction is among them. Such tracing can be implemented in a dynamic binary analysis tool like Pin or Valgrind. We tried both but we were not satisfied with the introduced running time overhead. Instead we chose to use debugger breakpoints to measure instruction coverage. Specifically we use GDB in its batch mode. For each live mutant, we first find the corresponding basic block in the original binary which we applied the instruction mutation in, and set a breakpoint at each of these addresses. Once the breakpoint is hit, it is safe to disable it in order to avoid any further overhead. After execution has finished, our tool parses GDB’s output to determine the set of breakpoints, and therefore corresponding mutants, which have been hit.

### V. EVALUATION

To evaluate our mutation analysis tool, we applied it to two sets of binaries. As an example of general-purpose binaries, we applied our tool to SPEC CPU 2006 benchmarks. As an example of a class of programs with more specific testing needs, we applied our tool to a collection of safety-critical embedded control binaries. For the purpose of the experiments

²https://github.com/s3team/uroboros
in this section we applied mutation operators to \texttt{Jcc}, \texttt{SETcc}, and \texttt{CMOVcc} instructions. \texttt{adc} and \texttt{sbb} are rarely used in SPEC binaries (they represent less than 5\% of mutants), and none of the embedded control binaries had those instructions.

A. SPEC CPU 2006

The SPEC CPU benchmark is a collection of CPU-intensive programs along with three different sets of inputs to stress compiler, system processor and memory. We took all 12 benchmarks written in C\(^{3}\) and applied our mutation analysis and ran them with the provided input sets. The benchmark input sets are not designed to maximize fault-finding, but the benchmark does carefully check the program outputs, for instance in case a program has been mis-compiled. When treated as functionality tests, these inputs can reveal a number of faults, so it is interesting to measure just how well they do.

For each benchmark, there are three sets of inputs which serve different purposes: \texttt{test} inputs are simple data to confirm the benchmark binary is functional, \texttt{train} inputs are used for feedback-directed optimization while building the binaries, and \texttt{ref} inputs are the actual data used for the performance test. All of these inputs are intended to work on valid benchmark binaries, meaning that if the generated output is different from what SPEC expects, a mismatch error is reported. We used each of these three input sets as the test suite for mutants and collected respective metrics.

We generated as many as possible mutants for each benchmark and then selected 1000 of them randomly, except for two benchmarks, \texttt{mcf} and \texttt{lbm}, where the maximum number of generated mutants were respectively 480 and 166 (due to smaller code size).

The results of running mutation analysis on the \texttt{test} input set is presented in Figure 5. Similarly Figure 6 shows the results for mutation analysis of the \texttt{train} input set, and Figure 7 presents the results on the \texttt{ref} input set. In all of these results, the benchmark ordering reflects decreasing size of code section from left to right. Live mutants are those that generated same output as the original benchmark binary. Killed mutants are those that failed to generate the expected output. A subset of the mutants are identified as trivial. Those are the mutants that fail to even start the execution due to some serious runtime error like a segmentation fault. For each benchmark we devised a trivial input which is described in Table II. Before passing the actual SPEC inputs, first the mutant is executed using these simple inputs; if it failed then it is a trivial mutant and is skipped for any further analysis.

Our mutation operator changes conditional control transfers, so a possible effect is to introduce an infinite loop in a mutant. In order to cover such cases, we took twice of the original binary runtime as the timeout. For our experiments, 10 minutes was the maximum required timeout. If a mutant’s execution runs out of time we consider it as a killed mutant.

The percentage of killed mutants (mutation score) with respect to each benchmark and input set is depicted in Figure 8. Again the benchmarks are ordered by code size. One evident result is that in general, the bigger input sets are able to kill more mutants. An interesting exception to this general observation is the \texttt{perlbench} case, where the \texttt{test} input set was able to kill more mutants. As described in the SPEC documentation, the \texttt{test} input for \texttt{perlbench} is derived from actual regression tests included with Perl (version 5.8.7). Smaller benchmarks also tended to have a higher mutation score. This may be because the larger programs have more complex and hard-to-test functionality.

To measure mutants reachability we post-processed the set of live mutants for each benchmark and input set. (We only need to check reachability of live mutants because a mutant must be reached to be killed.) A reachable live mutant is a case where the input data was not able to change the program state in a way that it affects the output. An example of such case is a conditional branch checking for a shortcut result. Such a branch has no visible effect on the output but just skipping some unnecessary steps. Figure 9 shows the percentage of live mutants that are reached for each benchmark and input set. As the results show, the live reachability is lower for larger binaries. This accounts in part for the related trend in mutation score, as even obtaining high coverage is more challenging in such binaries.

As a concrete example, we investigated a reachable live mutant for \texttt{gcc} on \texttt{test} input set. The mutation operator is applied to the condition at line 5 of Figure 10. As a result, the mutant binary has the instruction at line 4 of Figure 11 replaced by a \texttt{jmp} instruction. For the original binary, the

\footnotesize{\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\texttt{gcc} & \texttt{gobmk} & \texttt{lbm} & \texttt{milc} & \texttt{bzip2} & \texttt{hmmer} & \texttt{perlbench} & \texttt{spf} & \texttt{smp} & \texttt{sl} & \texttt{g++} \\
\hline
\hline
\end{tabular}}
TABLE II: Trivial Inputs for SPEC 2006 Binaries

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Trivial Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>C Compiler</td>
<td>An empty .i file</td>
</tr>
<tr>
<td>gobmk</td>
<td>Go game playing</td>
<td>An empty game (.sgf file)</td>
</tr>
<tr>
<td>perlbench</td>
<td>PERL Programming Language</td>
<td>An empty .pl file</td>
</tr>
<tr>
<td>h264ref</td>
<td>Video Compression</td>
<td>No input file</td>
</tr>
<tr>
<td>hmmer</td>
<td>Search Gene Sequence</td>
<td>-h</td>
</tr>
<tr>
<td>sphinx</td>
<td>Speech recognition</td>
<td>-h</td>
</tr>
<tr>
<td>sjeng</td>
<td>Artificial Intelligence: chess</td>
<td>Analyze a position to depth 0</td>
</tr>
<tr>
<td>milc</td>
<td>Physics: Quantum Chromodynamics</td>
<td>A grid of size 0</td>
</tr>
<tr>
<td>bzip2</td>
<td>Compression</td>
<td>An empty file, with compression level 0</td>
</tr>
<tr>
<td>libquantum</td>
<td>Physics: Quantum Computing</td>
<td>-h</td>
</tr>
<tr>
<td>lbm</td>
<td>Fluid Dynamics</td>
<td>Time step 0</td>
</tr>
<tr>
<td>mcf</td>
<td>Vehicle scheduling</td>
<td>0 trips</td>
</tr>
</tbody>
</table>

Fig. 7: Mutation results for the ref input set

Fig. 8: Mutation score for each benchmark and input set

Fig. 9: Percentage of reachable live mutants for each benchmark and input set

instruction at 0x804c2f1 is reached 26 times during the program execution where the branch is taken for 23 times: equivalent to the mutant’s behavior. For the other 3 times of fall-through, the next branch instruction at 0x804c2fc is always taken. This means the execution never reaches the instruction at 0x804c344 which is overwriting variable reversep with 0. This is an example where the reachability condition is met, but the necessity is not. Meaning that the mutation did not lead the execution to an incorrect state that can be detected by the test suite. This live mutant shows a lack of test adequacy in the test suite.

    static int
    noce_try_store_flag (if_info)
    struct noce_if_info *if_info; {
        int reversep;
        if (GET_CODE (if_info->b) == CONST_INT
            && INTVAL (if_info->b) == STORE_FLAG_VALUE
            && if_info->a == const0_rtx)
            reversep = 0;
    }

Fig. 10: An example of a reachable live mutant for gcc on test input, by applying mutation operator at line 5

Table III provides the mutation analysis results in more details. For each benchmark, the code section size is presented. For the number of trivial mutants, we counted the mutants that failed on inputs form table II. In addition, for each input set of test, train, and ref the results of mutation analysis are presented.
TABLE III: Mutation analysis results for SPEC 2006 binaries

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Code Size (in bytes)</th>
<th>Trivial Mutants</th>
<th>test</th>
<th>Mutation</th>
<th>ref</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>live</td>
<td>killed</td>
<td>live</td>
<td>reached</td>
<td>score</td>
</tr>
<tr>
<td>gcc</td>
<td>3316940</td>
<td>932</td>
<td>68</td>
<td>115</td>
<td>6.8%</td>
</tr>
<tr>
<td>gobmk</td>
<td>1492269</td>
<td>14</td>
<td>959</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>perlbench</td>
<td>953192</td>
<td>16</td>
<td>793</td>
<td>191</td>
<td>155</td>
</tr>
<tr>
<td>h264ref</td>
<td>498612</td>
<td>1</td>
<td>856</td>
<td>143</td>
<td>135</td>
</tr>
<tr>
<td>hmmer</td>
<td>251245</td>
<td>2</td>
<td>966</td>
<td>32</td>
<td>62</td>
</tr>
<tr>
<td>sphinx</td>
<td>183040</td>
<td>0</td>
<td>699</td>
<td>301</td>
<td>222</td>
</tr>
<tr>
<td>sseg</td>
<td>132224</td>
<td>19</td>
<td>843</td>
<td>138</td>
<td>53</td>
</tr>
<tr>
<td>mlc</td>
<td>117296</td>
<td>0</td>
<td>715</td>
<td>285</td>
<td>116</td>
</tr>
<tr>
<td>bzip2</td>
<td>569699</td>
<td>16</td>
<td>534</td>
<td>450</td>
<td>271</td>
</tr>
<tr>
<td>libquantum</td>
<td>39206</td>
<td>115</td>
<td>787</td>
<td>98</td>
<td>103</td>
</tr>
<tr>
<td>ibm</td>
<td>12322</td>
<td>9</td>
<td>104</td>
<td>53</td>
<td>19</td>
</tr>
<tr>
<td>mcf</td>
<td>12027</td>
<td>39</td>
<td>200</td>
<td>241</td>
<td>86</td>
</tr>
</tbody>
</table>

Fig. 11: Disassembly of code in Figure 10: the mutation is replacing instruction at line 4 with a jmp. But the execution state is not changed because either branch at line 4, or at line 9 is taken and the instruction at line 19 is never reached.

B. Embedded Control Binaries

Another set of binaries are control-intensive reactive programs for safety-critical embedded systems published by Byun et al. [11]. Cruise Controller is an automated throttle control software and Microwave is a control logic for an oven microwave, both developed by Collins Aerospace. Infusion Pump is a software for generic patient-controlled analgesia device developed by University of Minnesota. Docking Approach is an aerospace application developed by NASA. All the programs were initially modeled using Simulink/StateFlow [1], [2], translated to the Lustre programming language [20] to harness existing test automation tools, and finally translated to C using the Verimag Lustre V6 Tool Chain [3]. Microwave (C) is an exception; it had been manually coded in C using the StateFlow model as its specification. The binaries are compiled using GCC and with the -O2 optimization option. Since the size of the programs were smaller for control binaries, it was feasible to work with all possible mutants, between 98 and 3008 mutants per binary.

For these binaries, we used test suites automatically generated by Byun et al. using a coverage-guided test generation technique. The generated test suite is guaranteed by the back-end model checker to achieve the highest achievable branch coverage, condition coverage, and modified condition and decision coverage (MC/DC), and observable MC/DC [49] over the Lustre [20] source code. The high coverage does not guarantee the completeness of the test suite, but the test suite is at least good enough to exercise all the conditional constructs of the program and lead the program to produce an observably different output.

To apply our binary mutation tool, we executed all the generated mutants with all the tests, compared their outputs against the expected output of the original binary, and report the percentage of the mutants killed by the whole test suite. Acknowledging the ongoing debate on how to obtain an objective mutation score [30], [19], [25], we report the mutation score excluding trivial mutants. As we are generating test suites automatically, we consider a mutant to be trivial if it fails on all of the provided test inputs. That is because such mutants do not yield any information on the quality of the test suite. We also categorize the killed mutants into unique and duplicate mutants. Ideally, it is desirable for each mutant to uniquely represent a realistic fault that potentially exist in a program so that the mutation score can best capture the thoroughness of a given test suite.

Formally, we mark a mutant $p'$ as killed when there exists one or more test cases $t$ in our test suite $T$ that makes the mutant $p'$ produce an output different from the one of the original binary $p$: $\exists t \in T, p'(t) \neq p(t)$. Since it is computationally intractable or undecidable to precisely determine the equivalence and uniqueness among mutants, we rely on empirical evidence for categorization. A mutant $p'$ is trivial if $p'(t) = p(t)$ for $\forall t \in T$. A set of mutants $P'_\text{dup} \subseteq P'$ are deemed duplicate by subsets of the test $T_{\text{kill}} \subseteq T$ and $T'_{\text{kill}} = T - T_{\text{kill}}$ if $\forall t \in T_{\text{kill}}, \forall p' \in P'_\text{dup}, p'(t) \neq p(t)$ and $\forall t' \in T'_{\text{kill}}, p'(t') = p(t')$. In other words, a set of mutants that are killed by the exact same set of test cases and not killed by any other test case is deemed a duplicate set. This is an appropriate approach for these binaries because we had fine-grained suites with many small tests; we did not attempt a similar analysis for the SPEC mutants because their tests are more coarse-grained; they sometimes consist of as little as one test input per program.) For live (not killed) mutants, we did a reachability analysis through dynamic instrumentation with a Pin tool [27] to identify any mutant that was not killed because
the mutation point was never reached.

Finally, we compare the binary mutation score with the source-level mutation score as an indication that the two techniques are comparable. The utility of mutation analysis is mostly for assessing the adequacy of a given test suite. Since mutation analysis is mostly introduced as an artificial proxy for real faults to assess the utility of a test suite, a good set of mutants shall provide a good representation of real faults, which are typically assumed to be made by human developer in the source-code level [12]. This correlation, however, is hard to be established because of the difficulty of obtaining real faults in sufficient quantity. We instead performed a preliminary comparison of the binary-level mutation score to the source-level mutation score. We argue that if the binary-level mutation score is correlated to source-level mutation score, binary-level mutation can potentially be used in place of source-level mutation. We created a thousand source-level mutants by applying typical source-level mutation operators to the source code, proportionally to the relative percentage of the type of mutation opportunities available in the source code. For instance, we created more mutants for Boolean operators if a program has more Boolean operators.

Table IV presents the mutation score along with the number of mutants that fall into different categories. The mutation scores turned out to be relatively low for Infusion Pump and Docking Approach, which might make one suspect the quality of the binary mutants. However the source-level mutation scores were also relatively low, so our interpretation is that this difference instead is caused by features of these programs and inputs. Specifically we believe the low mutation scores for those systems are due to the extensive input validation logic inside the programs. For instance, a mutation can lead some inputs to corrupt the program state but the input validation logic can catch the flaw and produce an error code. Regardless of how the internal state is corrupted by different kind of mutation, the validation logic can mask the infection and produce the same output, leading to a low mutation score. We validated this observation by performing equivalence analysis with the source mutants using bounded model-checking in the Lustre source-level, where most of the unskilled mutants were equivalent to the source program for up to ten steps.

A direct comparison between the binary-level mutation score and the source-level mutation score is not possible because the types of mutation operators used for mutant creation were different between source-level and binary-level, and the number of mutants were not the same. However, we can still observe that the mutation scores are roughly correlated—the source-level mutation score is high when the binary-level mutation score is high. Although we cannot draw a statistically-grounded conclusion, this observation suggests that binary-level mutation analysis can be used in the place of source-level mutation when applicable, such as when source code is not available or doing source-level mutation analysis is expensive.

VI. DISCUSSION

While implementing our mutation analysis tool we encountered some compatibility and scalability limitations of the current Uroboros prototype, which seem consistent with ones described in more detail by the Rambltr authors [46]. We observed Uroboros’s performance to be sensitive to the compilation flags passed to build SPEC binaries, as for larger programs like gcc we were not able to use our own compiled benchmark under Uroboros to get a working disassembly; instead we used binaries provided with the Uroboros distribution for gcc and gobmk. (The Uroboros authors also provided us with the SPEC build configuration they used, though we have not yet reproduced their exact binaries.) In some cases where we got working disassembly, even a small change in the disassembly led to a nonworking binary, which we suspect indicates a symbolization failure, though due to time constraints we have not yet determined the detailed cause.

We also looked into using Rambltr, as embodied in the open-source angr and Patcherex tools, to take advantage of its improved symbolization, but though we had some success with binaries from the Cyber Grand Challenge, our initial attempts with small non-CGC binaries have not yet been successful. Potentially superset disassembly could be used for binary mutation (applying our mutation operator to every disassembly), but we have not explored this yet.

Another possible approach for creating binary mutants is to use a binary rewriting tool like Dyninst [43] to statically rewrite a binary to create mutations. Dyninst implements both CFG reconstruction and out-of-place instrumentation to address the challenges of binary rewriting. However our experience has been that a tool in which rewriting is controlled by a sophisticated API has a steeper learning curve: inserted instructions have to be expressed in terms of the API, and it may be that not all ways of using an API that initially sound applicable to one’s task are actually supported. Under the reassembleable disassembly approach, the “interface” to describe inserted instructions is more familiar, since one can just use standard assembly syntax. Generally speaking, there is an inherent challenge in static binary rewriting that one would always like to be solved in a reusable way; though reassembleable disassembly tools are not yet fully mature, our experience suggests that they provide a natural interface though which binary rewriting tools can be supported.

In this paper, we applied our mutation analysis on x86-32 binaries, but we believe our approach would apply without substantial changes for x86-64 as well: Uroboros (and Rambltr) already support 64-bit binaries, and the assembly-level representation of our mutated instructions would be very similar.

VII. FUTURE WORK

We plan to improve our binary mutation tool along a number of dimensions in the future. Perhaps the most important question yet to be answered is: how well can binary mutation represent real-world source-level bugs? We plan to explore the space of binary mutation operators and compare binary mutation with source-level mutation to find an answer to this question.

A reassembleable disassembly-based binary mutation approach is useful for replacing an instruction with one or more

---

4One symptom on the failure, a “Decoding Error” in the call to 
\texttt{cfg\_fast}, is the same as one mentioned in an issue at https://github.com/angr/patcherex/issues/20. Unfortunately, the solution mentioned there was not helpful in our case.
instructions of arbitrary size. But, we estimate that a substantial number of mutations can still be performed without increasing the size of a single mutated instruction. This class of mutations can be implemented with a simpler in-place rewriting that does not change the binary layout at all (if an instruction becomes shorter, the extra space can be filled with no-op instructions). We are interested to explore this approach as a comparison to the one used in this paper, to assess the trade-off between the benefits of simple rewriting and the set of mutations that are possible. Another important dimension to consider in binary mutation operators is whether they require replacing a sequence of multiple instructions, since safely replacing multiple instructions may depend on determining whether an intermediate instruction in the sequence could also be a jump target.

Exploring a large space of binary mutation operators on large binaries has the potential to create a large number of binary mutants. However, many of such mutants would be duplicates of each other or equivalent to the original program. We plan to explore techniques for identifying and preventing the creation of such mutants [26] in the future.

VIII. CONCLUSION

In this paper, we employed static binary rewriting via reassembling for analyzing software test suites. We described a class of mutation operators to generate mutants and depicted how building on reassemblable disassembly can make possible mutation analysis when source code is not available. We applied our mutation analysis tool on two sets of binaries: SPEC 2006 C benchmarks and embedded control binaries.

Our results from mutation analysis on embedded control binaries confirm that the applicability of binary mutation as the mutation scores is in alignment with source-level mutation scores. For the SPEC benchmarks our analysis revealed how the test inputs are not testing significant aspects of the code, and specifically for larger binaries, it is more challenging to have a comprehensive test suite. Thus our results show that binary mutation can be applied for the purpose of measuring test adequacy in the absence of source code.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Numbers 1563920 and 1514444. We also like to thank the Uroboros authors for helping us in reproducing their results.

REFERENCES


