Turning Programs against Each Other: High Coverage Fuzz-Testing using Binary-Code Mutation and Dynamic Slicing

Ulf Kargén Nahid Shahmehri

Department of Computer and Information Science
Linköping University
SE-58183 Linköping, Sweden
{ulf.kargen, nahid.shahmehri}@liu.se

ABSTRACT

Mutation-based fuzzing is a popular and widely employed black-box testing technique for finding security and robustness bugs in software. It owes much of its success to its simplicity; a well-formed seed input is mutated, e.g. through random bit-flipping, to produce test inputs. While reducing the need for human effort, and enabling security testing even of closed-source programs with undocumented input formats, the simplicity of mutation-based fuzzing comes at the cost of poor code coverage. Often millions of iterations are needed, and the results are highly dependent on configuration parameters and the choice of seed inputs.

In this paper we propose a novel method for automated generation of high-coverage test cases for robustness testing. Our method is based on the observation that, even for closed-source programs with proprietary input formats, an implementation that can generate well-formed inputs to the program is typically available. By systematically mutating the *program code* of such *generating programs*, we leverage information about the input format encoded in the generating program to produce high-coverage test inputs, capable of reaching deep states in the program under test. Our method works entirely at the machine-code level, enabling use-cases similar to traditional black-box fuzzing.

We have implemented the method in our tool *MutaGen*, and evaluated it on 7 popular Linux programs. We found that, for most programs, our method improves code coverage by one order of magnitude or more, compared to two well-known mutation-based fuzzers. We also found a total of 8 unique bugs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and debugging—*Testing tools*

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786844

General Terms

Security

Keywords

Fuzz testing, fuzzing, black-box, dynamic slicing, program mutation

1. INTRODUCTION

Memory safety errors, such as buffer overflows, are often the result of subtle programming mistakes, but can expose users to highly critical security risks. By supplying vulnerable software with specifically crafted inputs, attackers can often leverage these kinds of bugs to corrupt important data structures and execute arbitrary code with the privileges of the vulnerable program. Such attacks may lead to a complete compromise of the system on which the program is running.

Security bugs commonly stem from programs failing to appropriately handle rare corner cases during the processing of inputs. Traditional testing methods using manually written test cases, such as unit testing, are typically aimed at testing functional correctness of software under "normal" usage situations, and have therefore often proven insufficient to uncover critical security bugs. Intuitively, a security bug is often introduced as a consequence of a particularly unclear or complex part of the program specification. If test cases are based on the same specification, chances are high that the test writer has also failed to account for all obscure corner cases. In that perspective, automated test case generation, not biased by a specification or the developers interpretation thereof, is an attractive alternative for security testing. In this paper we consider *fuzz testing*, a practical and popular black-box technique for automatic test case generation.

The term fuzz testing, or simply *fuzzing*, was first coined by B.P. Miller in 1990 after noticing that, while working over a dial-up connection to a mainframe, random bit-errors in program inputs, introduced by interference from a thunderstorm, would often cause common UNIX utilities to crash. The subsequent study [20] showed that supplying random inputs to programs was a surprisingly effective means to discover robustness errors. More recently, fuzzing has gained widespread popularity as a security testing method. Many large software vendors, such as Adobe [7], Google [12], and

Microsoft [5] employ fuzzing as part of their quality assurance processes.

Most fuzz testing tools can be broadly divided into using either a *mutation-based* or a *generation-based* input-generation strategy [30]. The former entails performing some random transformations to an initial well-formed input, called a *seed*, while the generation-based approach uses a formal specification of the input format, e.g. a grammar, to generate inputs. While simple and easy to get started, mutation-based methods almost invariably yield lower code coverage than generation-based approaches. Mutated inputs often deviate too much from the format expected by the program, causing them to be rejected early in processing. However, as many input formats are highly complex, creating a formal specification for a generation-based fuzzer is often a very time-consuming and error-prone task. Furthermore, it is difficult to create tools and input-specification formalisms that can cater to all possible kinds of input formats. For that reason, security analysts are often forced to write their own target-specific fuzzers.

Another case where a security analyst may be confined to using a mutation-based approach is when testing third-party closed-source programs, as it is not uncommon for such programs to use proprietary or undocumented input formats. Analyzing closed-source programs for security flaws is a common (and profitable) endeavor in practice, as third-party independent security testing has become an important part in many software vendors' strategy to rid their programs of security bugs [13]. Several large companies, such as Google [1] and Microsoft [4], are e.g. offering substantial monetary awards for security bugs found in their products.

In this paper, we propose a novel method for completely automatic generation of high-coverage test inputs for robustness testing. Our method works directly at the machine-code level, and thus allows use cases where the analyst does not have access to source code or documentation of the input format. The method is based on the observation that, even in cases of proprietary input formats, the tester often has access to an implementation that can generate well-formed input to the program under test. Consider, for example, a hypothetical spreadsheet editor. Even if the editor is closed-source, and uses a proprietary format for spreadsheets, it can most likely store spreadsheets in a format that it can read itself. In a sense, the specification of the input format is *implicitly* encoded in the machine code of the program's output module.

On a high level, our method works by systematically introducing mutations (essentially bugs) into a *generating* program, and using the outputs from that mutated program as inputs to the program under test. Instead of simply mutating every instruction in the generating program, which would be impossible due to the undecidability of static disassembly, or using the naive approach of mutating every instruction observed to execute during one run, we use dynamic slicing to decide which instructions to mutate. This approach significantly reduces the number of necessary mutations (by 90%–95% in our experiments), with a corresponding reduction in time spent on executing mutants.

Our main hypothesis is:

By performing mutations on the generating program's machine code instead of directly on a well-formed input, the resulting test inputs are closer to the format expected by the program under test, and thus yield better code coverage.

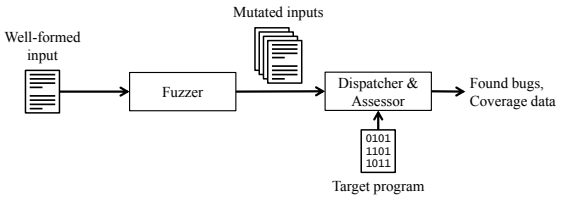

Figure 1: Typical mutation-based fuzzing pipeline.

We have implemented the proposed method in our prototype tool called *MutaGen*. We evaluated the method on several popular Linux programs, and found that, for most programs, it improved code coverage by one order of magnitude or more, compared to two well-known mutation-based fuzzers. In the 7 programs used in our tests, we found a total of 8 unique bugs.

2. BACKGROUND AND MOTIVATION

Typically, a testing campaign using mutation-based fuzzing proceeds as follows. First, a set of seed inputs of the appropriate type is collected. For example, if the objective is to test a PDF reader, a set of well-formed PDF files is compiled, either manually or e.g. using an automated web search. Each seed is then passed to the *fuzzer*, which performs mutations on well-formed inputs. Mutations can range from simply flipping bits at random, to more complex transformations, such as deleting, moving, or duplicating data elements. The fuzzing *dispatcher* is responsible for launching the program under test and supplying it with mutated inputs from the fuzzer. An *assessor* component is responsible for determining if the program handled the malformed input in a graceful way, or if it suffered a fault. When finding security bugs is the objective, a simple but effective approach is to simply observe whether the program under test crashes, as dangerous memory-corruption bugs typically manifest themselves as e.g. segmentation faults during fuzz testing. Since several different mutated inputs often trigger the same bug during a fuzzing campaign, the assessor commonly also has a means of estimating the uniqueness of found bugs. The industry-standard approach, used by most existing fuzzing tools, is to calculate a hash digest of the stack trace at the point of failure. The outcome of the fuzzing campaign is a set of *crashers*, mutated inputs that were observed to trigger unique faults. Figure 1 shows the testing pipeline for a mutation-based fuzzer.

As mutation-based fuzzing usually yields poor code coverage, the general view among security experts is that several million iterations are typically needed to find bugs in practice [24, 22]. Further, while seemingly simple to set up and get started, mutation-based fuzzing is highly dependent on the quality of the seed inputs [28], as well as various configuration parameters, such as the fraction of input bits that are mutated [16]. The traditional wisdom has therefore been that a great deal of tweaking, based on human intuition, is needed to achieve good results with fuzz-testing. Our method, by contrast, strives to eliminate the need for human effort as much as possible by leveraging information about the input format encoded in the generating program.

To illustrate the potential merits of our approach, consider for example a highly structured format, e.g. one based on XML. Blindly mutating a well-formed input is extremely

likely to break the structure of the input, causing it to be rejected early in processing by the program under test. By contrast, our method would cause the input mutation to be applied already on the internal, binary representation of the data, used by the generating program. By performing mutations before the final output encoding is applied, we hypothesize that our method is much better at maintaining the structure of the inputs, and thus reaching deeper states in the program under test.

3. DESIGN AND IMPLEMENTATION

Figure 2 shows a high-level overview of the MutaGen process. The dynamic slicer first executes the generating program on a given input, and computes a *cumulative dynamic slice* – a set containing instructions that contributed to computing at least one byte of the generator's output. That set is then fed to the *mutator*, which again executes the generating program, this time introducing mutations to the instructions contained in the slice. For each execution, one specific mutation (from a finite set of mutation operators) is introduced at one instruction, until all applicable mutations have been applied to each instruction in the slice. The resulting set of outputs from the generator, one per execution, is used as the set of test inputs to the program under test. The final stage is similar to that of a classical mutation-based fuzzer, using a dispatcher and assessor to run the program under test and detect failures. While our current implementation only considers input/output from files, it could also be extended to e.g. handle communication over network sockets.

Rather than finding all instructions that somehow influenced output, the purpose of our dynamic slicing is to compute a *conservative subset* of output-influencing instructions. More specifically, we only consider instructions that the output is directly data-dependent upon, and not implicit dependencies from indirect memory accesses or control dependence. The reasons for this are twofold. The first reason is that, in order to maintain output structure, we want to limit mutations to the *output generation* of the program, while keeping the program *semantics* largely the same. Therefore, we want to avoid performing mutations that e.g. significantly alter the control flow of generating programs. The second reason is that considering implicit dependencies almost invariably leads to overly-large slices. Previous work [29] has e.g. shown that considering implicit dependencies from indirect memory accesses in binary programs leads to many spurious transitive dependencies. Overly-large slices not only lead to more time spent on input-generation, but can also be detrimental to the overall stability of the input-generation process. If the selection of target instructions is too wide, chances are high that the generating program will at some point corrupt its environment, e.g. by overwriting an important configuration file, thus preventing future invocations of the program from running properly. Using a narrower selection of target instructions for mutation largely eliminates the need for isolation between mutant executions (e.g. by snapshotting and restoring the system), and is thus a key factor in reducing the runtime cost of our method.

The following subsections describe the MutaGen components in more detail.

3.1 Dynamic Slicing

We base our dynamic slicer on a system we developed in previous work [18]. In order to make the paper self-contained, we briefly describe the relevant details of that system here.

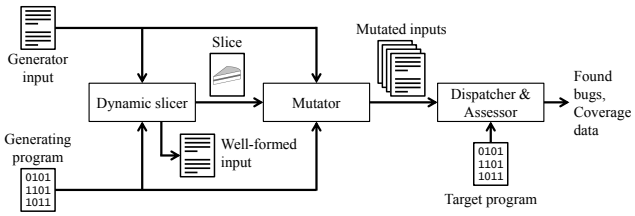

Figure 2: The MutaGen pipeline for test generation and execution.

The dynamic slicer is implemented using the Pin [19] dynamic binary instrumentation framework, and works directly on Linux x86 executable programs. First, the system records a *dynamic data dependence graph* (dyDDG), representing the entire data flow during one execution. That graph is then traversed backwards from a point of interest to compute a slice.

During execution, every instruction instance is assigned a unique ID, derived from its address and a per-basic-block execution count. These instruction instances effectively form nodes in the dyDDG. In order to record the data-flow of programs, a *shadow state* is maintained during execution, mirroring every register and memory location with a corresponding shadow register and shadow-memory location. The shadows are dynamically updated during runtime to always contain the ID of the instruction instance (i.e. dyDDG node) that last defined the corresponding real register or memory location. Each time an instruction is executed, the shadows of its input operands are inspected, so that edges to the preceding data-flow nodes can be recorded. Note that our system considers all executed instructions, including those in e.g. shared libraries.

In order to scale to realistic executions, the system uses secondary storage to record dyDDGs. Since dyDDGs are often too large to fit in RAM, this approach is necessary, but requires careful design of both algorithms and the graph representation, in order to maintain good performance despite high access-time delays.

In addition to its core function, we have modified the system to monitor all file-related system calls during execution of the generating program. Whenever the program writes a byte from memory to a file, the system makes a record of the node in the dyDDG that the written data corresponds to. This way, after the generating program has finished, we can compute the cumulative dynamic slice by simply finding all nodes that are reachable from any of the nodes corresponding to an output-byte. We perform a simple depth-first search, using the spanning-tree algorithm, to traverse the dyDDG and compute the slice. Thus, we don't need to re-visit already traversed nodes, ensuring good performance even for large output files.

One of the challenges with dynamic data-flow analysis of binaries is the lack of type information. When implementing similar analyses for high-level programs, one can use the program symbol table to allocate one shadow-variable for each variable in the program, ensuring accurate and precise data-flow tracking. In a binary program, however, the analysis engine only sees memory and register accesses. The most conservative approach would be to track dependencies

on the byte-level, i.e. allocating one shadow-byte for every byte of memory. However, this approach would be highly wasteful in terms of time and space overhead, since the vast majority of memory accesses use the native word size of the CPU (e.g. 4 bytes for the IA32/x86 architecture). Therefore, the slicer uses shadow memory with a minimum granularity of the native word size, a common approximation used by e.g. the popular dynamic-analysis tool Memcheck [25]. This approach may lead to lost data dependencies in case of sub-word writes, since an entry in shadow memory will always point to the instruction that last wrote any byte of the corresponding memory word. While such problems are rare in general, due to the scarcity of sub-word writes, the approximation can sometimes be problematic in the output-tracking stage of our analysis, since many programs emit single bytes as part of the final output encoding. To avoid having to track dependencies on the byte level, which would effectively quadruple the time and space overhead of slicing, we instead use the following method: When a sub-word write is observed during runtime, a *virtual* dependence edge to the previous definition of the full word is recorded, along with edges for the regular input operands. This way, no dependencies are ever lost, at the cost of many false dependencies. To compensate for false dependencies, we also modify the dyDDG traversal algorithm. If a virtual dependence edge is encountered during traversal, we only follow that edge if the corresponding defining instruction (i.e. the target of the edge) performs a sub-word memory write. This way, we prune the vast majority of spurious dependencies arising from re-used memory on e.g. the stack, while still accurately accounting for e.g. writes into byte-arrays. The remaining spurious dependencies may lead to slightly over-approximated slices, but does not seem to negatively affect our test-generation in practice.

3.2 Program Mutation

Program mutation has been studied extensively within the context of *mutation testing* [17]. The objective of mutation testing is often to determine the adequacy of test suites. To assess the test suite of a program, the tests are run on mutated variants of the program, referred to as *mutants*. By counting the number of mutants that were killed by the test suite (i.e. failed at least one test), the overall quality of the test suite can be quantified, and missing test cases can be identified. A mutant is created by applying a *mutation operator* to one or more statements in a program. A mutation operator applies a small syntactic change to the affected statement.

We have implemented our mutation component using Valgrind [26]. Valgrind performs dynamic binary instrumentation by first converting native instructions into the RISC-like VEX intermediate representation (IR). All instrumentation and analysis is performed on the VEX IR, which is transparently re-compiled to native code and executed by Valgrind, using a just-in-time compiler. We apply our mutations on the VEX IR, which avoids having to deal with the highly complex x86 instruction set, comprising more than 1000 different opcodes.

The code translation units used by Valgrind are single-entry-multiple-exit *superblocks* of VEX code. Complex instructions are broken down into several primitive VEX statements, and intermediate results are stored in single-static-assignment virtual registers called *temporaries*, which are local to one superblock. As an example, we consider the x86 instruction `mov eax,[ecx+edx*4]`, which calculates an address from registers `ecx` and `edx`, fetches a 32-bit word from that address, and writes the word to register `eax`. The corresponding translation into VEX is shown in figure 3a. The first two VEX statements fetch the contents of `ecx` and `edx` into temporaries `t0` and `t1`, respectively. The next two statements perform the address calculation, statement 5 reads a word from memory, and the final statement puts that value into `eax`.

```
1: t0 = GET:I32(4)
2: t1 = GET:I32(8)
3: t2 = Shl32(t1,0x2:I8)
4: t3 = Add32(t0,t2)
5: t4 = LDle:I32(t3)
6: PUT(0) = t4
```

(a) Original VEX IR

```
1: t0 = GET:I32(4)
2: t1 = GET:I32(8)
3: t5 = Shl32(t1,0x2:I8)
4: t2 = Sub32(t5,0x1:I32)
5: t3 = Add32(t0,t2)
6: t4 = LDle:I32(t3)
7: PUT(0) = t4
```

(b) VEX IR after applying mutation 2 to statement 3

Figure 3: Example of VEX IR for the x86 instruction `mov eax,[ecx+edx*4]`

Since mutation testing often aims to assess a test suite's ability to catch errors in the program logic, it is common to use mutation operators that replace arithmetic, relational, or logic operators, or the absolute values of constants. Recall, however, that in our setting the objective is to mutate the output generation of programs, while keeping program semantics mostly unchanged. Hence, most of our mutation operators perform small *arithmetic* changes to computations. We only mutate VEX statements that produce an intermediate result stored in a temporary. Mutating other statements, such as statement 6 in figure 3a, is redundant, since such statements only serve to propagate the result of a computation to a register or memory. Our 17 different mutation operators are summarized in table 1.

Mutations 1–4 perform simple arithmetic changes to the result of a computation, while mutations 7–10 and 11–14 perform the same changes to respectively the first and second operands of a statement, where applicable. Note that, since temporaries are single-static-assignment, we allocate a new temporary for the mutated input operand in these cases, ensuring that the mutation only affects the target statement. Figure 3b shows an example of applying mutation 2 (subtract 1 from destination) to statement 3 in the original VEX code.

Mutations 5 and 6 are special cases that are used instead of mutations 1–4 when the destination temporary is later directly used as an address. Statement 4 in figure 3a is an example of such a case. The rationale for treating these statements differently is that a multiplicative change of an

Table 1: Mutation operators used by MutaGen.

Mutation	Description
1	Add 1 to destination
2	Subtract 1 from destination
3	Multiply destination by 2
4	Divide destination by 2
5	Add 4 to destination
6	Subtract 4 from destination
7	Add 1 to first operand
8	Subtract 1 from first operand
9	Multiply first operand by 2
10	Divide first operand by 2
11	Add 1 to second operand
12	Subtract 1 from second operand
13	Multiply second operand by 2
14	Divide second operand by 2
15	Switch `and`/`or`
16	Switch `add`/`sub`
17	Switch signedness

address is extremely unlikely to result in a new valid address; the program would most likely only crash as a result of such a mutation. Further, it is much more likely that the modified address will point to a valid data object, e.g. an adjacent element of an array, if the native word size (4 bytes in x86) is used in the addition/subtraction, instead of 1.

Mutation 15 switches the bitwise And/Or operators for one another, where applicable, and mutation 16 equivalently switches the Add/Subtract operators. Finally, mutation 17 attempts to emulate a common programmer error where a signed data type is used instead of an unsigned, or vice versa. For operations that exist in both signed and unsigned variants, mutation 17 changes the signedness of the operation.

We systematically apply all applicable mutation operators to each VEX statement that is part of a translated x86 instruction in the dynamic slice. Using the terminology of mutation testing, we only perform first-order mutations, i.e. we only apply one mutation operator to one VEX statement per execution. Special-purpose instructions, such as floating-point or SIMD instructions, are ignored in our prototype implementation.

One of the challenges of mutation testing is to avoid *equivalent mutants*, i.e. mutants that are syntactically different but semantically equivalent. In this work, we redefine the term to mean *mutants that produce identical output given a specific input*, albeit being possibly semantically different. Equivalent mutants does not adversely affect the outcome of testing, but introduce unnecessary time overhead in the test generation. We perform basic data-flow analysis within superblocks to prune obvious cases that would lead to equivalent mutants. Performing mutation 3 on statement 1 in figure 3a is e.g. redundant if we know that `t0` is only used in statement 4, where mutation 9 is going to be applied anyway. Similarly, applying mutation 7 to statement 4 is redundant if mutation 1 has already been applied in an earlier execution. These optimizations reduce the number of equivalent mutants by roughly 30%. During test generation, we compute a hash of each output and keep only unique files.

3.3 Test Dispatching

The test dispatcher runs the program under test on each mutated file. To avoid missing crashes in cases where the program has registered handlers for certain signals, we perform simple instrumentation of the program to catch interesting signals (segmentation fault, illegal instruction, etc.) before they are sent to a signal handler. When a crash is detected, we make note of the input file that caused the crash, and, similarly to other fuzzing platforms, compute a hash digest of the stack trace at the time of failure, to distinguish between unique crashes.

4. EVALUATION

Similar to mutation-based fuzzers, our approach relies on simple syntactic mutations, without explicitly incorporating semantic information as in generation-based fuzzing. Therefore, an important question is what benefit our approach offers over mutation-based fuzzing. If there are no benefits, we would have a hard time justifying the increased complexity of our method to developers and security analysts.

Another important question is how effective our mutation operators are. Since, to the best of our knowledge, program mutation has not been applied in this context before, investigating the effectiveness and adequacy of the mutation operators is important, as it will help identify directions for future improvement of our method.

In the following section, we address these two questions, and present a preliminary evaluation of the overall performance and effectiveness of MutaGen.

4.1 Methodology

In our evaluation we used 6 different generating programs and 7 different file types. For each file type we generated two sets of mutated inputs, which were used to test a total of 7 popular Linux programs. All experiments were run on VirtualBox virtual machines with one virtual CPU-core per machine. All virtual machines were running Linux Mint 17.1. We used two Intel Xeon E3-1245 workstations at 3.3 GHz as hosts, each host running four concurrent virtual machines (one per host core).

All programs (generators, programs under test, or both) used in our experiments are listed in table 2. For the open-source programs, we used the latest versions supplied by the Linux Mint package manager. The latest version of the closed-source program `nconvert` was downloaded from the vendor's website. For our fuzzing targets, we have focused on programs that operate on popular document, image, or media types. Since such programs often take inputs originating from untrusted sources, such as the internet, they are popular targets for attackers. Fuzzing is therefore often applied as part of the quality-assurance process for these kinds of programs.

Table 3 shows the selection of generating programs. With the exception of `avconv`, we used two distinct inputs for each test generation run. For `avconv`, we used one input to generate mp3 and aac files using both constant and variable bit-rate encodings. All test sets were created by mutating the conversion of a sample file (column 2), using the configurations shown in column 3. The names of the resulting sets of test inputs are shown in column 4.

For each generator configuration we also performed one run without mutations, to produce a well-formed input. We

Table 2: Programs used in the experiments.

Program	Version	Description
avconv	9.16-6	Audio and video encoder/decoder. Open source.
convert	6.7.7-10	Commandline interface to the ImageMagick image editor/converter. Open source.
nconvert	6.17	Commandline interface to the XnView image editor/converter. Closed source.
pdftocairo	0.24.5	PDF conversion utility using poppler. Open source.
mudraw	1.3-2	PDF conversion utility based on mupdf. Open source.
pdftops	0.24.5	PDF to postscript conversion utility using poppler. Open source.
ps2pdf	9.10	Postscript to PDF conversion utility using ghostscript. Open source.
inkscape	0.48.4	SVG vector graphics editor. Open source.

refer to that input here as the *base case* of each generated test set.

Our selection of programs is limited in part by the fact that many popular Linux programs share the same code base. For example, there exist a plethora of PDF utilities and viewers for Linux, but the vast majority are based on one of `xpdf`, `poppler` (a fork of `xpdf`), `mupdf`, or `ghostscript`. Two programs that use the same library or code base are likely to crash on the same malformed input. To avoid potentially skewing the results of our testing, we have striven to choose only one testing target from each "family" of programs.

Table 4 shows the combinations of programs under test and input data sets used in the test runs. These combinations are henceforth referred to as *test configurations*. We measured instruction coverage for each test execution using a simple tool implemented with Pin. Since the coverage tool incurs a significant overhead, we also performed all runs without any instrumentation to get accurate time measurements.

To evaluate how our approach compares to existing fuzzing tools, we compare code coverage with two popular mutation-based fuzzers; `zzuf` [6] and `radamsa` [2]. `zzuf` performs simple random bit-flipping on inputs, while `radamsa` uses several input transformations known to have found bugs in the past, such as duplicating or moving data elements, incrementing bytes, removing bytes, etc. As seeds for the mutation-based fuzzers we used the base cases of each test set. Default parameters are used for both fuzzers. For each test configuration we ran the mutation-based fuzzers for as many iterations as there were files in the corresponding MutaGen test set. Note that this approach favors MutaGen, since the time taken for executing mutants is not accounted for. Therefore, we also ran `zzuf` and `radamsa` on all test configurations without coverage instrumentation for 24 hours, to allow a more fair comparison of the ability to find bugs.

4.2 Results

Program mutation. The results of our test-generation runs are shown in table 3. Column 7 and 8 show the total number of runs and files generated, respectively. Roughly 40%–80% of mutant executions results in an output. Column 9 shows the number of unique files produced for each configuration. With the exception of `inkscape`, about 10% of all mutant executions produce unique outputs. The number of mutation executions for `inkscape` is significantly larger than for the other programs, yet the number of unique files is much lower. The reason for this anomaly is unknown. One possibility is that the slicing step fails to identify "interesting" instructions to mutate for `inkscape`, e.g. because the output generation code makes heavy use of implicit dependencies (section 3).

Comparing the number of runs with the dynamic slice size (column 6) we see that each instruction results in roughly 10 mutations on average.

Benefit of dynamic slicing. Column 5 shows the total number of instructions executed at least once, for each generator configuration. The benefit of dynamic slicing is evident when comparing these figures with the slice sizes. Performing dynamic slicing reduces the number of instructions to mutate by between 90% and 95%, yielding a corresponding reduction in test generation time. The time required for computing slices was largely negligible in comparison to the time taken for executing mutants. The longest slice computation (`inkscape`, `sample-1.svg`), required 497.8 seconds, and the shortest (`nconvert`, `sample-1.tiff`) 7.6 seconds.

Coverage and found bugs. The outcome of our testing experiments is summarized in table 4. We report all coverage metrics as the relative cumulative increase in instruction coverage over the *base coverage*, i.e. the coverage when running the program under test on the base case of the test set. Note that we present coverage relative to the baseline only, as there is no clear definition of an absolute coverage ratio when testing binary programs that are dynamically linked.

With the exception of `nconvert` and `inkscape`, the coverage increase of MutaGen is typically one order of magnitude or more over that of `zzuf` and `radamsa`. `pdftops` particularly stands out with a coverage increase of over 150%. `inkscape` shows a more moderate 2.5x increase over the best mutation-based fuzzer, probably largely due to the low number of test inputs. The results also show that the coverage increase of the mutation-based fuzzers was much larger for `nconvert` than for the other programs. Upon closer examination, we found that the first iteration achieved a coverage increase of about 45%, and that for the remaining iterations the increase was negligible. We speculate that this may be due to some particularly complex error-management code being run when certain malformed inputs are encountered.

Column 6 shows the time required to run tests without coverage instrumentation. Due to the small number of test cases, no test configuration requires more than 30 minutes to complete.

As mentioned in section 4.1, measuring coverage of the mutation-based fuzzers for only as many iterations as there are files in the corresponding MutaGen test set may not offer a fair comparison. Figure 4 shows the coverage of `zzuf` and `radamsa` versus iteration for 10 test configurations. (Due to space constraints we only show one plot per program and file type, but all configurations show similar results.) The "X" in the plots marks the last iteration when any coverage improvement was observed. With the possible exception of `inkscape`, all cases appear saturated already after a small number of iterations. It is therefore unlikely that running the fuzzers for a longer time would produce significantly

Table 3: Mutation configurations and results.

Generating program	Input	Configuration	Test set name	Total ins.	Slice size	# runs	# files	Uniq. files	Time (hours)
avconv	sample-1.mp3	mp3 → mp3, CBR	mp3-cbr	108,279	5,358	51,692	34,892	3,841	24.46
		mp3 → mp3, VBR	mp3-vbr	108,783	5,339	51,411	35,254	4,383	25.08
		mp3 → aac, CBR	aac-cbr	103,567	5,337	51,125	34,401	5,507	25.78
		mp3 → aac, VBR	aac-vbr	103,580	5,371	51,689	34,918	5,716	27.71
convert	sample-1.png	png → tiff	tiff-1	64,957	5,312	46,328	23,108	5,008	15.51
	sample-2.png		tiff-2	64,627	4,164	39,440	20,530	3,037	11.55
nconvert	sample-1.tiff	tiff → png	png-1	32,385	3,055	33,446	20,815	5,275	6.50
	sample-2.tiff		png-2	30,104	2,017	22,152	14,355	3,041	3.63
pdftocairo	sample-1.pdf	pdf → pdf	pdf-1	104,328	8,353	81,262	66,868	9,714	33.99
	sample-2.pdf		pdf-2	129,349	10,320	97,759	80,277	10,715	44.74
pdftops	sample-1.pdf	pdf → ps	ps-1	56,120	4,823	44,964	32,934	5,124	13.17
	sample-2.pdf		ps-2	76,870	5,883	53,001	40,835	4,890	26.32
inkscape	sample-1.svg	svg → svg	svg-1	235,372	13,996	121,719	44,478	978	102.08
	sample-2.svg		svg-2	236,187	14,299	123,864	44,798	1,067	105.94

Table 4: Testing results.

Program	Test set	Base cov. (# ins.)	MutaGen				zzuf	radamsa
			Crashes	Unique bugs	Time (s)	Rel. cov. increase	Rel. cov. increase	Rel. cov. increase
ps2pdf	ps-1	179,819	1	1	1319.2	48.19%	0.85%	0.92%
	ps-2	181,282	0		1734.8	42.87%	0.85%	0.89%
pdftops	pdf-1	54,775	5	3	1783.3	154.23%	2.38%	1.82%
	pdf-2	52,890	3		1177.1	165.10%	2.44%	1.83%
mudraw	pdf-1	63,103	0	0	987.8	58.86%	2.06%	3.07%
	pdf-2	69,066	0		736.0	57.95%	2.26%	2.94%
inkscape	svg-1	238,259	0	0	280.9	6.36%	2.61%	0.74%
	svg-2	239,250	0		312.4	6.72%	2.64%	2.60%
avconv	mp3-cbr	108,568	0	0	459.9	25.46%	3.68%	0.68%
	mp3-vbr	109,124	0		496.8	12.98%	1.25%	0.64%
	aac-cbr	110,587	0		518.6	22.03%	0.25%	0.40%
	aac-vbr	110,677	0		545.5	20.29%	0.30%	0.53%
convert	png-1	64,542	0	1	195.9	50.80%	3.79%	4.52%
	png-2	64,043	0		108.0	21.72%	3.85%	3.85%
	tiff-1	65,918	0		338.6	44.58%	4.00%	4.21%
	tiff-2	64,020	1		154.1	45.73%	4.20%	4.40%
nconvert	png-1	29,332	0	3	161.1	90.52%	45.57%	2.03%
	png-2	29,134	0		55.6	70.35%	45.23%	45.28%
	tiff-1	32,491	2		292.3	96.93%	41.51%	41.67%
	tiff-2	31,979	4		135.8	93.91%	42.40%	42.41%

higher coverage. Again, `inkscape` stands out somewhat. In order to be consistent with the methodology used for the other programs, only about 1000 iterations were used for `inkscape`, which is likely too few to achieve full saturation.

Some of the plots show conspicuous similarities between the results for `zzuf` and `radamsa`. We suspect that this may be due to the naming scheme used for generated files. We add the iteration number as a suffix to each file name, which causes the length of the file name to change deterministically. For `avconc-aac-cbr` we e.g. see small jumps in coverage at the 10th, 100th, and 1000th iteration.

MutaGen produces 16 inputs that crash programs, out of which 8 were unique bugs according to the stack trace. Some of the bugs are simple correctness errors, such as divisions by zero. Other bugs involve trying to free memory at invalid addresses, which could have security implications. Neither the short instrumented runs, nor the 24-hour uninstrumented runs of `zzuf` or `radamsa` yielded any bugs.

Effectiveness of mutation operators. Table 5 summarizes the fractions of mutants and test inputs contributed by each mutation operator, for all test sets. Some mutation operators are significantly more common than others. For example, every VEX statement that is candidate for mutation has an output operand, which means that mutations 1–4 are always applied, while for example mutation 15 (switch And/Or) is very uncommon. Note that the number of mutants for mutations 1–4 are not exactly equal. Since we identify the candidate VEX statements online during the test generation phase, we may in some rare cases fail to apply a mutation when the preceding mutant crashed very early. This results in a very small variation in the number of mutants for each "family" of mutation operators. We do not expect this to affect our results in a significant way.

Due to our pruning of equivalent mutants the "add" and "subtract" mutations 7 and 8, as well as 11 and 12, are rare in comparison to their "multiply" and "divide" counterparts (section 3.2). We also note that mutation 16 (switch Add/Subtract) is very common, probably much due to the commonality of address-generation expressions like the one in figure 3.

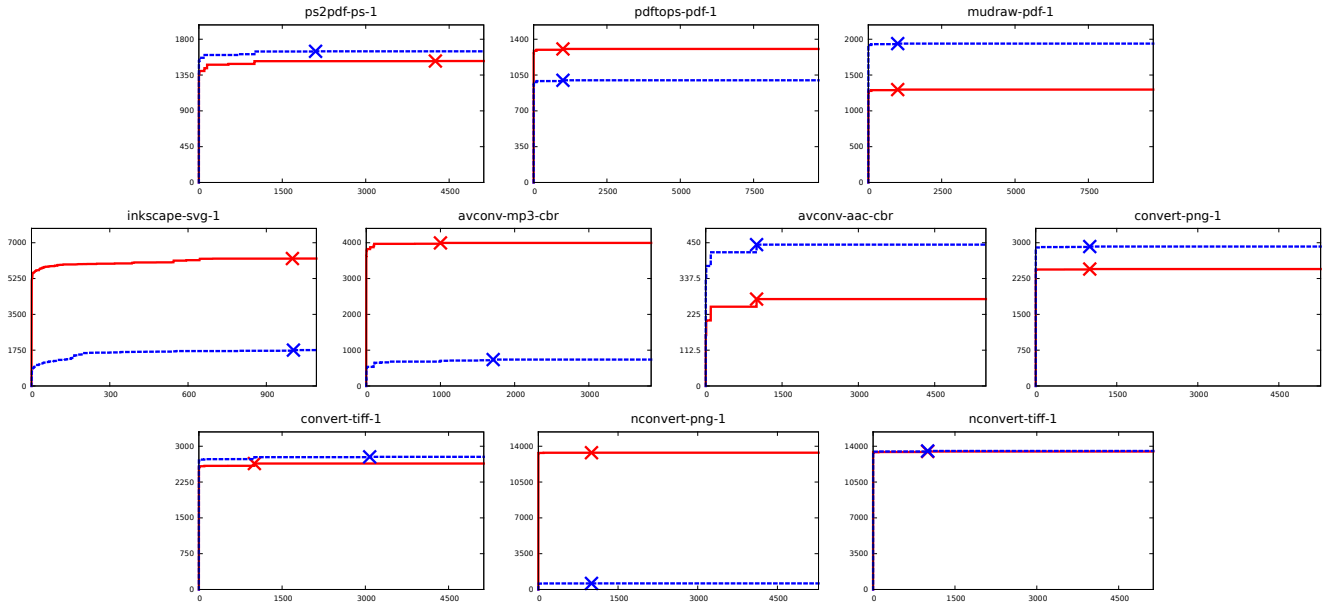

Figure 4: Instruction coverage versus iteration for the mutation-based fuzzers. The solid red line represents `zzuf`, while the dashed blue line represents `radamsa`.

Within the field of mutation testing, it has previously [27] been shown that, out of a larger set of mutation operators, a small number of operators can achieve results very close to the full set. We wanted to investigate if the same was also true in our context. To this end, we investigated the instruction coverage achieved by the optimal set O_n of mutation operators of size n, for each n between 1 and 17. Since this is an instance of the minimum set-cover problem, which is known to be NP-complete, we simply computed our results by means of exhaustive enumeration of all combinations. For each combination of n mutation operators, we calculated the average of the relative coverage increase over all test configurations, and selected the combination with the highest average coverage. We also counted how many of the bugs would be found with the optimal reduced operator set of each size.

At first, it may seem appropriate to weight the coverage of each mutation operator by e.g. the number of test cases, or the number of mutants, since table 5 shows that some mutations are much more common than others. However, applying such linear weighting would unduly favor the rare mutations, due to the very pronounced saturation effect observed in random testing. Consider for example two mutation operators m_i and m_j, yielding the respective sets of test inputs T_i and T_j, $|T_i| > |T_j|$. These test sets in turn cover instructions I_i and I_j when used as inputs to some program. If we randomly sample $|T_j|$ inputs from T_i to produce a smaller test set T_r, that test set will still yield a coverage that is significantly higher than $\left(|T_j|/|T_i|\right)\cdot|I_i|$, due to said saturation effect. For this reason, we do not apply any weighting of coverage scores when computing optimal set combinations.

The optimal operator sets of each size are presented in table 6. Using only 6 mutation operators achieves 97.6% of the mean coverage of all operators, and finds 7 of the 8 bugs in our experiments. This corresponds to a 53% reduction of mutant executions (cf. table 5). With 10 operators, we achieve 99.6% of the mean coverage and find all bugs, while achieving a 26% reduction of mutant executions. The most effective operator is number 2, which alone achieves 73% of the total mean coverage. Operator 15 is the least effective, which is perhaps not surprising, since it is also the least common one.

Several interesting observations regarding the effectiveness of different types of mutation operators can also be made from the results:

- *Subtraction is better than addition.* All the "subtract 1" operators (2,8,12) perform better than the "add 1" operators (1,7,11). This is likely because subtraction can cause larger semantic modifications to inputs by changing a small number to a very large number by means of a wrap-around. As Boolean variables are commonly represented by the integers 0 and 1, subtraction can also change truth values in both directions ($0 \rightarrow 2^{32}$, $1 \rightarrow 0$), while addition can only change false to true ($0 \rightarrow 1$).

- *Operators 1–4 largely subsume operators 7–10.* Mutations on destination operands appear to largely subsume mutations on the first input operand. This is not an unexpected result, since many VEX statements that have only one input operand simply serve to receive some intermediate result produced by an earlier statement. Note that we prune the trivial cases where the result of a computation is only used once within a superblock (section 3.2).

- *Mutating the second operand is effective.* Mutations 13,14 are significantly more effective than 9,10. We speculate that this is because, in expressions with constants, the second operand often holds the constant value. As constants are not first loaded into temporaries in the VEX IR, they are not subjected to mutation by operators 1–4. Mutating the second operand

Table 5: Contributions from each mutation operator.

Mutation	Mutants	Test inputs	Crashes	Bugs
1	7.38%	11.30%	3	1
2	7.35%	10.87%	2	2
3	7.34%	8.26%	0	0
4	7.34%	8.89%	0	0
5	8.65%	9.98%	3	3
6	8.63%	7.31%	0	0
7	1.62%	2.70%	0	0
8	1.60%	3.04%	1	1
9	9.51%	4.05%	0	0
10	9.36%	4.91%	0	0
11	1.28%	0.82%	0	0
12	1.28%	1.34%	0	0
13	9.03%	9.09%	2	1
14	9.03%	7.91%	3	1
15	0.44%	0.32%	0	0
16	8.07%	8.41%	2	1
17	0.36%	0.78%	0	0

Table 6: Effectiveness of reduced mutation operator sets.

Optimal mutation operator set	Cover	Bugs
2	73.116%	2/8
2 5	84.132%	5/8
2 5 13	90.139%	6/8
2 4 5 13	93.744%	6/8
2 3 4 5 13	96.118%	6/8
1 2 3 4 5 13	97.610%	7/8
1 2 3 4 5 13 16	98.561%	7/8
1 2 3 4 5 6 13 16	99.193%	7/8
1 2 3 4 5 6 13 14 16	99.450%	7/8
1 2 3 4 5 6 8 13 14 16	99.585%	8/8
1 2 3 4 5 6 8 10 13 14 16	99.702%	8/8
1 2 3 4 5 6 8 10 12 13 14 16	99.805%	8/8
1 2 3 4 5 6 8 9 10 12 13 14 16	99.881%	8/8
1 2 3 4 5 6 8 9 10 12 13 14 16 17	99.936%	8/8
1 2 3 4 5 6 7 8 9 10 12 13 14 16 17	99.970%	8/8
1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17	99.993%	8/8
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17	100.000%	8/8

could therefore be important for instructions that use constants.

- *Mutating address-generators is worthwhile.* Since mutations 5 and 6 are only applied on statements that compute addresses, we can implicitly draw the conclusion that mutating address-generating instructions is effective, both for improving coverage and for finding bugs. One reason that addition beats subtraction here could be that pointers to arrays are common in programs. Adding 4 to such a pointer will shift it to the second element of the array, while subtracting from it will cause it to point outside the array.

5. DISCUSSION AND FUTURE WORK

The results in section 4.2 clearly suggest that our main hypothesis holds; mutating a correct implementation of a program that can generate well-formed inputs achieves a considerably better coverage and finds more bugs, compared to traditional mutation-based fuzzing. In this section, we further analyze the results and suggest directions for future work.

Comparison to mutation-based fuzzing. While it is difficult to fit MutaGen into one of the traditional fuzzing paradigms, we believe its typical use cases to be similar to those of mutation-based fuzzing. For example, MutaGen could potentially serve as a "drop-in" replacement for a mutation-based fuzzer when e.g. crafting a formal input specification for a generation-based fuzzer is not feasible. Compared to previous results from mutation-based fuzzing [22, 24, 28], our method has a relatively high crash density, i.e. ratio of crashes to test cases, and has a very high bug density per crash (50%). That suggests that MutaGen can produce a small set of *high-quality* test inputs, which can find bugs in a small number of test runs. While test generation is fairly time-consuming with our current implementation, MutaGen can generate a set of inputs that can be used to quickly test several programs that accept the same file type. This is not possible with traditional mutation-based fuzzing, where the main cost lies in executing the program under test on a very large number of test inputs. Further, while the results of our limited evaluation are not statistically significant, the fact that MutaGen found several bugs, while `zzuf` and `radamsa` found none, may suggest that our approach is better at finding complex bugs that mutation-based fuzzing fails to catch.

Comparison to generation-based fuzzing. Our method also bears some resemblance to generation-based fuzzing, most notably the ability to exploit information about input formats to generate semi-valid inputs. There are, however, several important differences. First, our method is potentially more versatile, as it does not require a formal specification of an input format. For example, the well-known generation-based fuzzer SPIKE [3] uses a block-based approach to define input formats. While SPIKE has proven very successful at testing e.g. implementations of binary network protocols, it would not work for inputs that are not easily fitted into a block-based formalism. Secondly, a key difference between mutation-based and generation-based methods is the latter's ability to test long-running programs with state, such as implementations of stateful network protocols, by modeling the protocol state machine. As our method relies entirely on implicit information about the structure of input, such models cannot be incorporated into our approach. An interesting direction for future work is to investigate if MutaGen can sufficiently retain the semantics of the generating program, so that its state machine is kept "synchronized" with the server when testing stateful protocols. Finally, apart from a formal specification of the input format, generation-based fuzzers typically also need a set of heuristics for generating semi-valid inputs. Such heuristics are often compiled from inputs or input-patterns that have been known to trigger bugs in the past [30, 15]. Our method does not currently allow incorporating such heuristics when applying mutations.

The general consensus among security experts is that methods based on generation typically achieve better coverage than those based on mutation. Miller and Peterson [23] e.g. report 76% higher coverage for generation compared to mutation when fuzzing a PNG parsing library. For this reason, a comparison between our approach and generation-based fuzzing would be interesting. There are several practical complications, however, that make such a comparison diffi-

cult. For example, to the best of our knowledge, there are no publicly available robust generation-based fuzzers available for many of the popular input formats that we use in our evaluation. Should we repeat our evaluation, comparing against generation-based fuzzers instead of mutation-based, we would be forced to write our own fuzzers for many of the input formats. Aside from the significant extra work effort, this would introduce an obvious risk of bias. A direct side-by-side comparison may also be difficult due to the aforementioned differences in use cases and application areas. Analogous to mutation-based fuzzing, the results of a comparison are e.g. likely to be very dependent on the choice of seed-inputs for the generating programs.

Input space compression. A traditional mutation-based fuzzer, such as `zzuf`, has a search space of size 2^k for a seed of size k bits. The obvious benefit of a generation-based fuzzer, as well as our approach, is that the search space can be reduced by applying information about the input format. In order to cover a sufficient part of the input space, a mutation-based fuzzer also needs a large and diverse selection of seed inputs. The optimal selection of seeds is still an active area of research [28]. With our approach, the equivalent problem is to identify appropriate generating programs, as well as inputs and configuration parameters for these programs. Investigating the importance of the choice of generating programs is an important topic for future work. Another interesting question is whether the code coverage of the *generating* program is a good estimate of the covered input space.

Performance improvement. One obvious direction of future work is to improve the performance of MutaGen. Currently, the most time consuming part of the MutaGen pipeline is the execution of mutants to produce test cases. The current mutation tool is implemented with Valgrind, which facilitates rapid development, since it is not necessary to deal directly with the intricacies of the x86 instruction set. However, Valgrind does incur an overhead of at least 10x. A more performance-conscious choice would be to perform mutations by means of e.g. binary-patching, which would allow mutants to execute at near-native speed.

Mutant reduction. Another way to reduce the cost of test generation is to reduce the number of mutants. Our investigation of the effectiveness of mutation operators shows that, in our particular experiments, we could reduce the number of mutants by about 50% while still finding most bugs and retaining high coverage. We intend to further study the potential for mutant reduction in future work. We also intend to verify if the reason that mutations of the second operand perform so well is indeed due to arithmetic with constants. If that is the case, an alternative mutation strategy could be to only perform mutations 1–5, along with direct manipulation of constants.

Intermittent mutation. Finally, another strategy for input generation is to only intermittently inject errors into the generating program, rather than statically mutating instructions. This could potentially retain the semantics of the generating program to a larger degree, possibly enabling even higher coverage of the program under test. Similar to mutation-based fuzzing, however, the number of generated inputs would be unbounded.

6. RELATED WORK

Following the pioneering work by Miller et al. [20, 21], many of the early advances in fuzzing were made outside of academia. Recently, Woo et al. [32] studied scheduling algorithms for mutation-based fuzzing, and Rebert et al. [28] investigated the importance of seed selection. While both of these works focus on traditional mutation-based fuzzing, MutaGen could still benefit from their findings. Efficient scheduling algorithms could help MutaGen prioritize which parts of a generating program to mutate first, if there is a fixed time budget for executing mutants. Our approach could also benefit from the findings of Rebert et al. when selecting inputs to generating programs.

Generation-based fuzzers have proven particularly effective at finding bugs in compilers and interpreters. Yang et al. use generation-based fuzzing to find bugs in C compilers [33], and Holler et al. use grammars, in combination with code fragments known to have previously triggered bugs, to test interpreters [15]. Due to the highly stringent requirements on input structure, and the general scarcity of input-generating programs, we believe generation-based fuzzing to be favorable over our method in this specific domain.

Symbolic execution is an alternative to black-box testing methods. The SAGE [14] and Mayhem [9] systems can for example generate high-coverage test cases using concolic execution of program binaries. While such methods have seen tremendous development during recent years, their scalability is still limited.

Wang et al. propose a hybrid approach [31] to improve the coverage of programs that perform integrity checks on their inputs. A constraint solver is used to repair checksum fields in fuzzed files, allowing them to pass integrity checks. The authors note that the checksum-repair stage is time consuming, and that their method would fail to handle input formats that use cryptographically strong integrity checks. Our approach, by contrast, sidesteps that problem altogether by allowing mutations to be applied before checksum computations.

Several previous works describe methods for automatically recovering input formats by dynamic analysis of program binaries [8, 11, 10]. The recovered input formats could later be used to drive a generation-based fuzzer. While similar in spirit to our work, these methods are limited by the fact that input formats can only be recovered with respect to a pre-determined formalism, e.g. context-free grammars. For that reason, they face a similar problem as generation-based fuzzing systems; there is no one formalism suitable for describing all input formats. Our method tackles the problem of input generation at a lower level, and does not need to make assumptions about properties of input formats.

7. CONCLUSION

In this paper we have presented a novel method for automatic test case generation using dynamic slicing and program mutation. We empirically evaluated the method on 7 Linux programs and found that it improved code coverage with at least one order of magnitude in most cases, compared to two well-known fuzzers. We found a total of 16 crashing inputs, 8 of which represent unique bugs. We evaluated the effectiveness of our mutation operators, and suggested ways to further improve the coverage and performance of our method.

8. REFERENCES

[1] Chrome reward program rules. http://www.google.com/about/appsecurity/chrome-rewards/.

[2] A crash course to radamsa. https://code.google.com/p/ouspg/wiki/Radamsa.

[3] Immunity inc - resources. http://www.immunityinc.com/resources/index.html.

[4] Microsoft bounty programs. https://technet.microsoft.com/en-us/library/dn425036.aspx.

[5] Sdl process: Verification. http://www.microsoft.com/security/sdl/process/verification.aspx.

[6] zzuf - multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf.

[7] B. Arkin. Adobe reader and acrobat security initiative. http://blogs.adobe.com/security/2009/05/adobe_reader_and_acrobat_secur.html, 2009.

[8] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 317–329, 2007.

[9] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[10] P. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 110–125, 2009.

[11] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 391–402, 2008.

[12] C. Evans, M. Moore, and T. Ormandy. Google online security blog – fuzzing at scale. http://googleonlinesecurity.blogspot.se/2011/08/fuzzing-at-scale.html, 2011.

[13] M. Finifter, D. Akhawe, and D. Wagner. An empirical study of vulnerability rewards programs. In *Proceedings of the 22nd USENIX Security Symposium*, pages 273–288, 2013.

[14] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[15] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[16] A. D. Householder and J. M. Foote. Probability-based parameter selection for black-box fuzz testing. Technical report, CERT, 2012.

[17] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[18] U. Kargén and N. Shahmehri. Efficient utilization of secondary storage for scalable dynamic slicing. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 155–164, 2014.

[19] 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. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[20] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[21] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin-Madison, Computer Sciences Department, 1995.

[22] C. Miller. Babysitting an army of monkeys. CanSecWest, 2010.

[23] C. Miller and Z. N. Peterson. Analysis of mutation and generation-based fuzzing. Technical report, Independent Security Evaluators, 2007.

[24] D. Molnar and L. Opstad. Effective fuzzing strategies. CERT vulnerability discovery workshop, 2010.

[25] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 65–74, 2007.

[26] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[27] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, 1996.

[28] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the 23rd USENIX Security Symposium*, pages 861–875, 2014.

[29] A. Slowinska and H. Bos. Pointless tainting?: Evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 61–74, 2009.

[30] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[31] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 497–512, 2010.

[32] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 511–522, 2013.

[33] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294, 2011.