Application of Domain-aware Binary Fuzzing to Aid Android Virtual Machine Testing

Stephen Kyle Hugh Leather
Björn Franke
University of Edinburgh
s.kyle@ed.ac.uk, hleather@inf.ed.ac.uk,
bfranke@inf.ed.ac.uk

Dave Butcher
Stuart Monteith
ARM Ltd.
dave.butcher@arm.com,
stuart.monteith@arm.com

Abstract

The development of a new application virtual machine (VM), like the creation of any complex piece of software, is a bug-prone process. In version 5.0, the widely-used Android operating system has changed from the Dalvik VM to the newly-developed ART VM to execute Android applications. As new iterations of this VM are released, how can the developers aim to reduce the number of potentially security-threatening bugs that make it into the final product? In this paper we combine domain-aware binary fuzzing and differential testing to produce DEXFUZZ, a tool that exploits the presence of multiple modes of execution within a VM to test for defects. These modes of execution include the interpreter and a runtime that executes ahead-of-time compiled code. We find and present a number of bugs in the in-development version of ART in the Android Open Source Project. We also assess DEXFUZZ's ability to highlight defects in the experimental version of ART released in the previous version of Android, 4.4, finding 189 crashing programs and 15 divergent programs that indicate defects after only 5,000 attempts.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—testing tools; D.3.4 [*Programming Languages*]: Processors—run-time environments, compilers

General Terms virtual machines; testing; reliability; security

Keywords testing; compiler testing; virtual machine testing; fuzzing; Android; DEX; ART; random testing

VEE '15, March 14–15, 2015, Istanbul, Turkey..
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3450-1/15/03. . . $15.00.
http://dx.doi.org/10.1145/2731186.2731198

1. Introduction

The creation of a new application virtual machine (VM) is a complicated task and, like the development of any piece of complex software, is bound to create bugs during the process. Once the initial development is complete, there is the potential for new features and performance improvements to introduce bugs as well. VMs may typically have multiple methods of executing a supplied program, such as an interpreter or using a just-in-time (JIT) compiler. The developers may wish to optimize such methods of execution further, or even add new ones. When this happens, how can the developers easily test that new bugs are not being introduced? While scrupulous developers will always aim to cover every corner case, it must be assumed that bugs will inevitably make their way into software.

In this paper, we will focus on the development of the new ART runtime that is used to execute Android applications. The Android mobile operating system continues to enjoy increasing popularity, recently reaching over a billion active users measured over a 30-day period. It is therefore important to use rigorous testing to find and remove as many defects that were introduced during development as possible, before the new software is released to the public.

Test suites are a good way to catch bugs during development, but are typically limited in size, and cannot easily capture all possible interactions between code optimizations in a compiler. The unit test suite of the ART runtime we discuss in this paper currently stands at around 200 tests, taken and extended from the test suite of the original VM that Android previously used. Meanwhile, GCC's unit test suite numbers over 100,000 tests, although GCC has been in development for over 25 years. How can we get from 200 to 100,000 test cases without waiting for 25 years of bug reports? Indeed, regardless of size, these test cases are usually created when bugs are found, and are only intended to prevent regressions. How do we test for bugs that we are not aware of?

When multiple modes of execution are available within a VM, differential testing can be used, where all modes of execution are given the same program, and are expected to

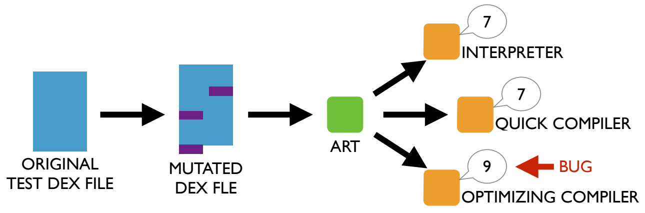

Figure 1. An overview of differential fuzz testing for VMs. A DEX program is run using multiple methods of execution, and in this example, one of the methods—the *optimizing* compiler—reveals a defect in the way it has compiled and executed the program.

form a consensus about how the program should have been executed. With a VM that is supposed to provide a single platform and set of semantics regardless of the underlying execution platform, this is a sound expectation. Figure 1 visualises how we can use this to test for bugs in the VM. As there is no single canonical reference specification for the ART VM, we assume that the conceptually simplest (and so typically slowest) mode can be considered to be the reference mode, as it will not contain many performance-increasing optimizations that have the potential to introduce incorrect behaviour.

Another common approach to testing is fuzzing, creating random test cases through either *generative* means, where a new test is produced from no initial seed, or *mutative*, where a test is mutated from a seed program. The combination of fuzzing and differential testing, while applied to domains such as C compilers in the past[12, 16], seems a good candidate for testing a VM with the desired rigor that Android's VM would need.

We present DEXFUZZ, a tool that takes an existing test suite and performs mutative fuzzing in order to produce new tests, in an attempt to explore the boundaries of the VM. These tests are executed by the various methods of execution of ART, such as the completed *quick* compiler, the interpreter, and the in-development *optimizing* compiler. In the rest of this paper, we will refer to these modes of execution as "backends" of the VM. We use differences in how these backends execute the mutated tests to identify bugs within in the VM. Some of these bugs are present in the common verification phase of the runtime, while others have been found in the compilation phase. In this paper, we present examples of how these bugs were found, how they came to be, and how they were fixed.

We ran DEXFUZZ on the experimental version of ART released in Android KitKat (v4.4). Even after only 5,000 iterations from one seed program, we had produced 189 programs that crashed the VM and 15 programs that led to divergent behaviour. We checked and confirmed that one of the divergent programs arose from the same bug that we later patched in ART, as described in Section 3.1.1. If DEXFUZZ had been available during the initial development of ART, it might have been possible to find and fix these issues earlier.

```
 1 // The array-length bytecode reads the length
 2 // field of an array object, 8 bytes offset
 3 // from the start of the object.
 4 //
 5 // Original DEX code:
 6 // v0 = undef, v1 = undef, v2 = array
 7 const/4 v0, 8
 8 array-length v1, v2
 9 // v0 = 8, v1 = array.length (LD [array, 8])
10 //   v2 = array
11
12 // Mutate array-length from v2 to v0:
13 const/4 v0, 8
14 array-length v1, v0
15 // v0 = 8, v1 = ??? (LD [8, 8]), v2 = array
```

Listing 1. An example of a DEX mutation leading to a compilation bug, due to incorrect bytecode verification. The compiler will always produce code that reads from the provided array pointer plus 8 bytes. By mutating the array-length instruction to take v0 as an input, we now illegally read from address $8+8=16$.

1.1 Motivating Example

The ART VM is a new implementation of the "Dalvik VM" that executes register-based DEX bytecode on Android. In Figure 1, we see an example of DEX bytecode that loads the constant 8 into virtual register v0 at line 7. On line 8, the *array-length* instruction then reads into v1 the length of the array whose reference is stored in v2. If we mutate the *array-length* bytecode to instead read from whatever "reference" is stored in v0, then ART's bytecode verifier should reject this bytecode, because v0 does not contain an array reference at this point. However, the verifier previously failed to do this. The compiler backend for ART would then assume that it was safe to produce code that reads from *(v0+8), and so the resulting code allowed arbitrary reading of the VM process' address space, if the constant loaded into v0 was also modified. With the error checking done by the verifier reducing the complexity of the compiler, this requires that verification be sound, which was not always the case, and was a common source of issues.

The Java language aims to provide a "security sandbox" for any program that executes within it. It is still possible to construct pointers and read memory arbitrarily if a mechanism such as the Java Native Interface (JNI) is used, so more trust can be placed in an application being "well-behaved" if it contains no JNI code. Therefore such an application may come under less scrutiny by security analysts, while a piece of crafted DEX code as presented in Figure 1 could be used in combination with other exploits to read private data. With the open model of Android, and ease with which applications can be accepted into the various application stores that use Android, there is a significant possibility of encountering accidentally or even maliciously malformed DEX files.

Using DEXFUZZ, we have found a number of bugs during the development of ART, and have submitted patches to the Android Open Source Project (AOSP) code base where ART is developed. We focussed our efforts on finding and reporting bugs in the *quick* compiler, as this is the default mode of execution for ART. It is our expectation that DEXFUZZ will become more useful as the *optimizing* compiler becomes more sophisticated. As the compiler surpasses the complexity of the *quick* compiler, optimizations are likely to introduce subtle bugs as they are initially developed. We hope that use of this tool may help prevent these bugs ever being put into released versions of the ART VM.

1.2 Contributions

The contributions presented in this paper are the following:

1. the description of domain-aware binary fuzzing for DEX bytecode, improving over *i.* naive approaches that have a greatly reduced chance of ever producing valid programs; *ii.* approaches that may only produce valid programs; *iii.* approaches that are limited to only source code generation, and therefore potentially fail to test as many bytecode sequences in a VM.
2. an application of differential testing to the ART virtual machine in Android, that exploits the multiple execution methods available in the VM to find bugs using mutated test programs.
3. a presentation of bugs found in the Android Open Source Project using these approaches, and how they were fixed.

1.3 Overview

In Section 2, we present an overview of our fuzzing and testing strategy for finding bugs in the ART VM. In Section 3 we present an analysis of the mutation process, as well as giving examples of some bugs that were found using our system and how they were found. In Section 4 we provide some discussion of the impact of our system, and look at related work in the field in Section 5. Finally, we consider potential future work for DEXFUZZ in Section 6 and conclude in Section 7.

2. DEXFUZZ

Fuzz testing as a concept initially referred to the random creation of input to test the capabilities of a program or Application Programming Interface (API), particularly to test its ability to gracefully handle erroneous input. Fuzz testing is typically divided into two categories: *generative*, and *mutative*, where generative does not require a seed to generate a new piece of input, while mutative does.

2.1 Naive fuzz testing

The most basic form of mutative fuzzing is to take some seed input and randomly flip bits in order to produce some new, mutated input. We could apply this technique immediately to producing test programs for the VM, but this isn't likely to yield very useful results. The ART VM is supposed to provide a secure sandbox for program execution, and so it must verify any bytecode that it is expected to execute. This verification forms a hierarchy of checks that range from checking that the two input registers to an `add-float` bytecode actually contain float values, to calculating an Adler-32 checksum on the file, and checking this against a provided checksum in the header. Because of these checks, it is extremely likely that any bytecode that is produced through random bitflips will fail some part of verification, if not at least the checksum that protects the entire DEX file.

We tested this claim by producing a simple fuzzer that fuzzed a test program a million different ways for three different fuzzing strategies. In all cases, the DEX file header is untouched, and then the Adler-32 checksum is recalculated after fuzzing, to ensure that we are checking that such simple fuzzing will be rejected by some other aspect of verification aside from the checksum. We ran each program through the DEX code viewing tool (*dexdump*) as this reports when a DEX file has basic structural errors.

The first strategy iterated through every byte after the header, with a 50% chance of having its value varied by +/- 30. This produced no programs that passed structural verification, out of a million programs. The second strategy changed the chance to 1%, and still no programs passed structural verification. The final strategy used a chance of 0.1%. This strategy did produce programs that passed structural verification - 14%. With further testing, 4% passed full verification when executed with ART, and of those that passed, 97% ran successfully and 0.14% showed divergent behaviour between different ART backends - 0.006% of the million programs that were generated. Focussed mutation could achieve a much higher rate of divergence than this - we stated in Section 1 that we were able to find 15 divergent programs in only 5,000 attempts, compared to the rate of 60 in 1,000,000 seen here. Additionally, this approach doesn't have the potential to insert or delete instructions, or insert even more complex constructs such as new methods or classes.

Typical generative approaches to fuzz testing for compilers have focussed on producing valid programs in the relevant source language. In the case of Csmith[16], this desire for valid program generation stemmed from the presence of undefined behaviour with certain sequences of C code. With a bytecode format like DEX, the concept of undefined behaviour does not exist. Either a sequence of bytecode has a well-defined set of semantics, or the sequence must not verify. Therefore, we are not concerned with the threat of undefined behaviour, and would actually prefer to generate some invalid programs, in a bid to ensure the verifier of ART is robust. Additionally, prior generative approaches have found it difficult to produce a system that fully utilises all features of the language that they are generating code for[16]. Using a set of programs from a test suite that should at the very least contain all features of the language as a seed basis for testing seems a better approach.

Neither of the above strategies are ideal for fuzzing bytecode, so in DEXFUZZ we adopt an approach that applies some degree of intelligence to the use of mutations.

2.2 Mutating DEX Bytecode

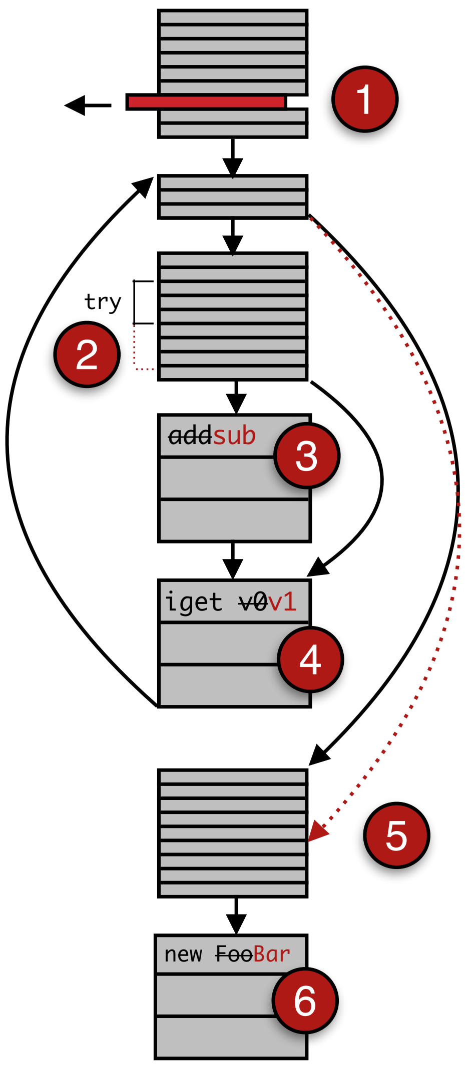

Figure 2. Some examples of DEX bytecode mutations applied to a method formed of six basic blocks. In ①, an instruction is deleted. In ②, the boundaries of a try block are expanded. In ③, an add operation is replaced with a subtract operation, operating on the same types and virtual registers. In ④, the virtual register specified by an *iget* instruction is changed. In ⑤, the branch from the 2nd basic block to the 6th is shifted forward by a few instructions. Finally, in ⑥, a new instance of an object of class Bar is created instead of Foo.

Our fuzzing process can be characterised as domain-aware mutation of bytecode. Figure 2 provides an example of some mutations performed within a single method. Our hope is that the application of a small number of relatively simple, but domain-aware, mutations can lead to both programs that verify, but also highlight bugs in the ART VM through unexpected handling of mutated bytecode.

```
parse DEX file into mutatable methods;
$Methods$ = set of mutatable methods;
$Mutators$ = set of mutators;
$methodCount$ = random(2,10);
for $i$ in $methodCount$ do
    $method$ = selectRandom($Methods$);
    $mutationCount$ = random(1,3);
    while $mutationCount > 0$ do
        $mutator$ = selectRandom($Mutators$);
        if $mutator.canMutate(method)$ then
            $method$ = $mutator$.applyMutation($method$);
            $mutationCount$ = $mutationCount$ - 1;
        else
            if reached mutation attempt threshold then
                give up mutating method;
            end
        end
    end
end
write methods and mutated methods into new DEX file;
```

Algorithm 1: Mutative fuzzing process for DEX files.

Algorithm 1 shows pseudo code for the mutation process. DEXFUZZ will parse a program's DEX file and produce a set of mutatable code items, each of which represent a method within the program. In order to increase the likelihood that the program will successfully verify, DEXFUZZ limits the number of methods that will be mutated to a random value between 2 and 10. A random subset of the available mutatable methods are then selected using this value, and each in turn is mutated. Each method has between 1 and 3 mutations applied to it by default, although this value can be configured. DEXFUZZ randomly selects from the list of available mutators that are listed in Table 1. Because each mutator checks and reports if it is able to mutate the given method (for example, BranchShifter will only mutate methods containing branches), this process repeats until the desired number of mutations has been applied, or a maximum number of attempts is reached. After all mutations have been applied to the correct number of methods, a new, mutated DEX file is produced by DEXFUZZ.

Why does this mutation strategy improve over simple bit-flipping fuzzing? At the very least, this allows us to add and remove instructions within the method. Additionally, we can ensure that branches always remain within the code area, type or field pool indices fall within the range of types of fields available in the DEX file, and a specified virtual register in an instruction is actually allocated in the given method. These are what we consider to be simple structural constraints that the verifier can easily check for.

Why does it improve over generative or mutative fuzzing that only produces legal programs? While we have highlighted some verifier constraints as simple, others are more complicated, such as checking the types of values in virtual registers are valid wherever they are used. It is important to test these constraints, and so we do not provide any guarantee that a mutation will leave the bytecode in a completely legal state. Indeed a few of the bugs we have found in ART were found in the verifier, and this was made possible by allowing the production of invalid programs.

2.3 Differential Testing of Multiple VM Backends

Test suite programs are typically used to test for VM correctness by executing the program, and comparing its output against what it is expected to produce. However, with the generation of new test programs, the expected output is unknown. Therefore, we rely on the fact that the multiple methods of execution available to a VM are intended to produce the same result—one of the design goals of many VMs, but particularly ones related to executing Java code—to test our mutated programs.

Figure 1 visualises our differential testing strategy. We can run mutated programs using the *quick* and *optimizing* compilers, as well as the interpreter. Additionally, in our search for bugs, we ran mutated programs on both 32 and 64-bit variants of the VM. When they fail to reach a consensus about the output of the program, it is highly likely that one of the modes of execution has discovered a bug.

3. Results

We have used DEXFUZZ to find defects in the in-development version of ART when running on an ARMv8 platform, in both AArch32 and AArch64 execution modes. The version of DEXFUZZ we have used to find bugs is designed to send programs to the target platform via the Android Debug Bridge (ADB) for execution. Because of the overhead involved in sending programs to the platform, we verify all mutated programs on the host machine first using the DEX compiler, `dex2oat`, that has been compiled for the host machine, and only upload programs that pass this initial test. The host and target versions of `dex2oat` verify all DEX files the same way.

3.1 Finding Bugs

In this section we present some examples of bugs that we have found in the ART VM in AOSP, and explain both how they were discovered by DEXFUZZ, and what caused their presence. Some bugs we report were found in the verifier. Although we do not have two verifiers to perform differential testing with, the use of such testing can still lead to bugs being found in the verifier. This typically occurred when a hole in the verifier would lead to the *quick* compiler and interpreter producing different results for the falsely verified code. This was usually because the *quick* compiler would make assumptions about the code it could produce based on properties of the DEX bytecode that the verifier had allegedly proven.

3.1.1 Reading the instance field of a non-reference bearing virtual register

In DEX bytecode, there exist instructions such as *iget*, that allow the reading and writing of instance fields of objects. The verifier of ART checks that the data currently in input virtual registers for a given bytecode have the correct types, with respect to the types the bytecode operates on.

```
const/4 v2, 1
iget v0, v1, MyClass.counter
add-int/2addr v0, v2
iput v0, v1, MyClass.counter
```

This piece of DEX bytecode represents the increment of the *counter* field of a MyClass object whose reference resides in *v1*. When mutated by the VirtualRegisterChanger mutation, which changed the *v1* on line 2 to *v2*, the verifier accepted this bytecode, despite *iget* now attempting to read the instance field of the constant 1 in *v2* rather than any reference to an object. The *quick* compiler would then assume that it was safe to emit native code that loads from the address stored in the input to *iget*, plus the offset of counter. If the constant loaded into *v2* initially was modified, then memory could be arbitrarily read from the process' address space. We submitted a patch for the verifier to AOSP that ensured this check was performed.

3.1.2 Mixed float/int constant usage causes arguments to be passed incorrectly

While many arithmetic and logical operations in DEX bytecode are aware of the types of data they are using, other bytecodes such as data movement and constant loading instructions are not. Consider the following code sample.

```
const v0, 1
const v1, 1.0
invoke-static {v0}, void Main.doInteger(int)
invoke-static {v1}, void Main.doFloat(float)
```

This code loads two constants, 1 and 1.0, and passes them to methods that take integer and float arguments, respectively. The *const* bytecode is typeless, and just loads a bit pattern into a virtual register. As such, the load of the float value 1.0, and the integer 0x3f800000 (the IEEE 754 representation of 1.0) are indistinguishable in bytecode. This requires that type inference be performed - the compiler must look at the uses of constants to determine their types. Because *v1* is used as a float at line 4, it must be a float. If the VirtualRegisterChanger mutation changes the register on line 3 to *v1* instead of *v0*, then the constant 1.0 in *v1* is now being passed to both methods. Type inference will determine

Type	**Description**	**Example**
BranchShifter	Change the target of a branch by a small delta.	*if-eqz v0, +05 → if-eqz v0,* ***+07***
ComparisonBiasChanger	Change the bias of a comparison operation.	*cmpg-double v0, v1 → cmp**l**-double v0, v1*
ConstantValueChanger	Change a constant used for a constant load, or an immediate in an arithmetic operation.	*add-int/lit8 v0, v1, 7 → add-int/lit8 v0, v1,* ***18***
InstructionDeleter	Delete a bytecode.	*const v0, 2; const v1, 4 → const v0, 2*
InstructionDuplicator	Duplicate a bytecode.	*or-int v0, v0, v2 → or-int v0, v0, v2;* ***or-int v0, v0, v2***
InstructionSwapper	Swap two bytecodes.	*monitor-enter v3; move v4, v2 →* ***move v4, v2; monitor-enter v3***
OperationChanger	Change the arithmetic or logical operation performed by a bytecode, preserving types.	*add-int/2addr v0, v1 →* ***div****-int/2addr v0, v1*
PoolIndexChanger	Change the index into a type/method/field pool used by a bytecode.	*new-instance v2, type@007 → new-instance v2,* ***type@023***
RandomInstructionGenerator	Generate a random bytecode, with random operands, and insert it into a random location within the method.	*nop; nop → nop;* ***throw v6****; nop*
TryBlockShifter	Move the boundaries of a try block.	*TRY { move v0, v1; } move v2, v6 → TRY { move v0, v1; move v2, v6* ***}***
VirtualRegisterChanger	Change one of the virtual registers specified by a bytecode.	*iget v1, v2, field@015 → iget v1,* ***v0****, field@015*

Table 1. A complete listing of all the DEX mutations currently performed by DEXFUZZ.

that *v1* contains both a float and an integer value, and this is legal according to DEX bytecode specifications.

This lead to a miscompilation in the *quick* backend of ART, however, because the code that passes arguments to the method using the appropriate calling convention would rely on this type-inference information, resulting in problems in an environment where integer and floating point (FP) values are passed in separate physical registers. Because the mutated register *v1* had been marked as a float type at line 3, the emitted native code would pass this argument in a FP physical register, when the callee method would expect to find its integer argument in a core (non-FP) register. The callee method would then use whatever data happened to be in that core register, and execute incorrectly. We have submitted a patch to AOSP that changes the calling convention compiler code to look at the specified types of the callee method instead of type-inference information.

3.1.3 Compiler crash from an unreachable check-cast

Verification of DEX bytecode is only performed on reachable code - code flow analysis takes place during verification, reporting errors like execution reaching the end without an explicit *return* bytecode. It is not intrinsically illegal to have unreachable bytecode, however. Consider the following DEX bytecode.

```
return-void
check-cast v0, V
return-void
```

The *check-cast* instruction is illegal when considered alone - it checks that it is acceptable to cast the value in *v0* to a *void* type, or throw a `ClassCastException` otherwise. *Void* is never a legal type to cast to, but because the *check-cast* is not reachable it is never rejected by the verifier. For most bytecode, this would not lead to any problems, however the *quick* backend performed a later optimization where it scanned across DEX bytecode it was to compile, to find *check-cast* instructions that it could elide because they would never throw the exception. This optimization would reach the unreachable check-cast and then cause the compiler to crash when an assertion was broken. While we have presented the simplest example of this bug here, this was found when the RandomInstructionGenerator mutation inserted such an invalid *check-cast* after a *return* statement. We submitted a patch for AOSP that makes this scan only consider instructions that were flagged as visited during verification.

3.1.4 Verifier allows jump to move-result

In DEX bytecode, every method invocation instruction that calls a result-returning method must be followed by a *move-result* bytecode to save the result. This means that verification must check that *move-result* always comes immediately after an invoke. Consider the following DEX bytecode.

```
002: if-nez v0, 008
004: invoke-static {}, int Main.getInteger()
007: move-result v1
008: add-int v0, v0, v1
```

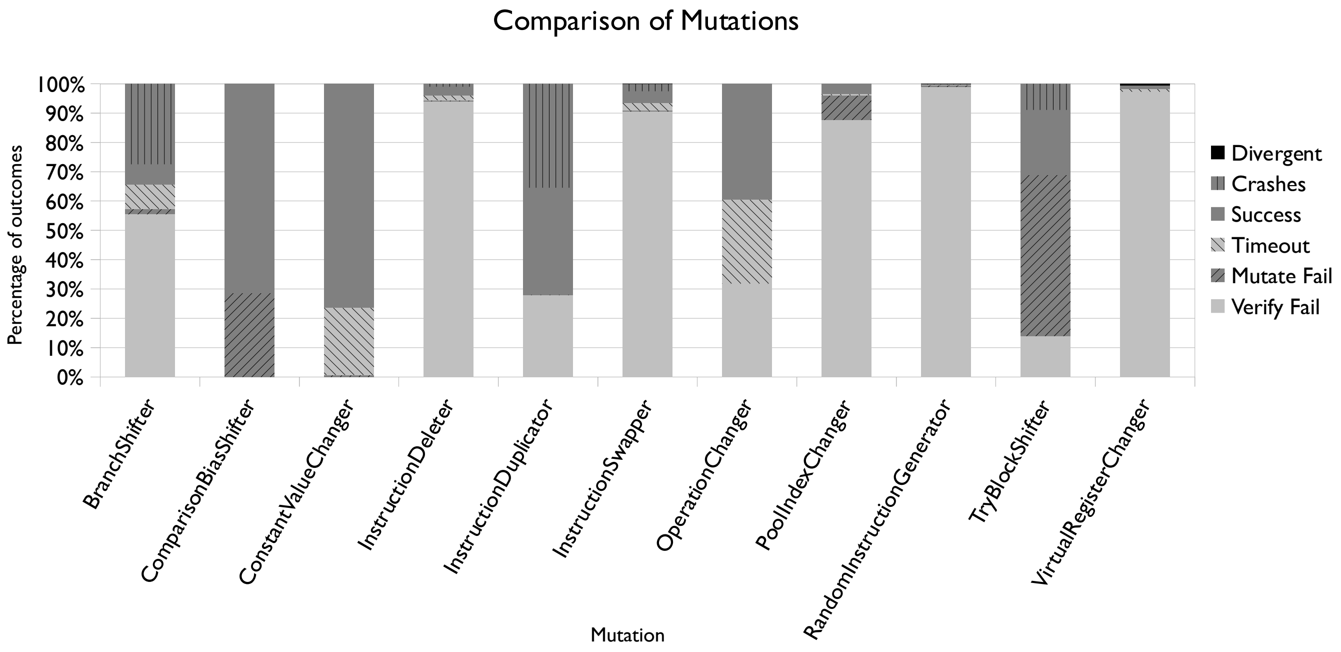

Figure 3. Breakdown of resulting programs created by individual bytecode mutations. In each case, we have run DEXFUZZ for 2,000 iterations, and report what percentage of these programs failed to be mutated, failed to verify, timed out, crashed, were successful, or produced program divergence, when run on the experimental version of ART available in Android KitKat (v4.4).

Mutation	Verify Fail	Mutate Fail	Timeout	Success	Crashes	Divergent
BranchShifter	1110	37	167	137	549	0
ComparisonBiasShifter	0	572	0	1428	0	0
ConstantValueChanger	0	13	459	1528	0	0
InstructionDeleter	1880	8	32	60	20	0
InstructionDuplicator	558	5	0	728	709	0
InstructionSwapper	1811	7	52	79	50	1
OperationChanger	636	2	571	791	0	0
PoolIndexChanger	1753	169	7	71	0	0
RandomInstructionGenerator	1979	10	2	5	2	2
TryBlockShifter	279	1098	0	444	179	0
VirtualRegisterChanger	1944	2	18	25	0	11

Table 2. Absolute counts of execution outcomes for each mutation shown in Figure 3.

We can see that the *move-result* instruction legally immediately succeeds the *invoke-static* instruction. For *move-result*, the *quick* compiler will typically emit native code that moves data from the return register of the platform's calling conventions into whatever storage location is associated with the specified virtual register, be it another physical register, or a stack location. For example on the ARM platform, the data will be read from the `r0` register.

The verifier failed to check if there were any instructions that branched directly to a *move-result* instruction, however. If the BranchShifter mutation made the *if-nez* instruction jump to the *move-result* instruction instead of *add-int*, then we could have a case where we execute the *if-nez* instruction, and then the *move-result* instruction, although we have not returned from an invoke. Previously, the verifier would accept this, and invalid native code would be produced. In this case, if the branch was taken, then execution would jump directly to the copy of the `r0` register, although it would no longer be known at this point what data `r0` might contain. We submitted a patch to AOSP that ensures any form of branching directly to a *move-result* instruction is rejected.

3.2 Characterization of mutations

We have presented 11 mutations that can be applied to DEX bytecode that aim to highlight bugs within the ART VM. In this section, we characterise which mutations are more responsible for creating verified programs and finding program divergences. Because we have already used DEXFUZZ to find and patch a number of defects within ART, in this section all test programs were executed with the experimental version of ART released in Android KitKat (v4.4), where we are certain that there are bug-indicative divergences to be found. We use a single test program as a seed in these experiments—which was confirmed to work in this version of ART prior to mutation—in order to be sure we are evaluating the characteristics of mutation.

We ran DEXFUZZ 11 times, each time enabling only one of the available DEX mutations. For each mutation, we produced 2,000 candidate programs, and we report how many verified, how many failed to mutate at all, how many passed verification and ran successfully on the platform without divergence, how many crashed, and how many produced divergent behaviour that may indicate the presence of a defect. The differential testing we use in these experiments compares only the output of the interpreter and the compiler in this older version of ART.

In Figure 3 and Table 2, we present the breakdown of execution outcomes for the different mutations we perform. Some are clearly more successful at finding divergent behaviour than others, with the *VirtualRegisterChanger* most responsible for producing divergences. 97% of the programs generated by this mutation failed verification however, so this clearly demonstrates the necessity of allowing the generation of potentially illegal programs in order to find divergences. The other mutations that found divergences were *RandomInstructionGenerator*, and *InstructionSwapper*, which also generated a large percentage of programs that failed to verify, at 99% and 91%, respectively. A number of these divergent programs arose from the bug described in Section 3.1.1.

Many mutations were responsible for producing crashes in ART, with the *InstructionDuplicator* and *BranchShifter* mutations most likely to lead to a crash. From a cursory analysis of the backtraces of these crashes, it appeared that ART had problems with SSA transformation during the compilation stage, which may have been exacerbated by the variety of unique control flow graphs produced by the *BranchShifter* mutation in particular.

Looking at each mutation in turn, *BranchShifter* produced invalid programs for roughly half of its iterations, and was otherwise responsible for a large number of crashes. The *ComparisonBiasShifter* mutation either produced successful programs, or failed to mutate the program at all, due to maximum attempt thresholds mentioned in the description of DEXFUZZ's operation. *ConstantValueChanger* never produced any invalid programs, but created a number of long-running programs, presumably when changes to constants meant loops ran longer than the allowed timeout. *InstructionDeleter* was another mutation that led to a large percentage of invalid programs, but was also able to find a few crashes.

InstructionDuplicator had a roughly three-way split in terms of crashing, successful and invalid programs produced, with no divergences found, while *InstructionSwapper* found a few crashes, but mainly produced invalid programs. Despite *ConstantValueChanger* only changing an operation with a particular set of input types to an operation with the same set of input types, 30% of its programs failed to verify, contrary to expectation. This was traced to bytecodes that operated on 64-bit integers like *add-long* being changed to shift operations on longs, such as *ushr-long*, where the second input, the shift amount, is actually expected to be a 32-bit value, rather than 64-bit.

PoolIndexChanger resulted in a lot of invalid programs, as for example, method invocations were changed to call new methods with incompatible argument types. As might be expected, *RandomInstructionGenerator* lead to the largest percentage of invalid programs, but did also find crashes and divergences. The *TryBlockShifter* mutation found a large number of crashes too, which lends weight to the idea that control flow issues were a major source of crashes in the compiler at the time. Finally, as stated above, *VirtualRegisterChanger* was the most successful mutation, finding the largest number of divergences, but failed to find any crashes at all.

3.3 Effect of multiple mutations

We have seen that some mutations are capable of finding divergent behaviour on their own, and indeed some have been individually responsible for many of the bugs that we have discovered in ART so far. With the use of these small mutations there is a hope that a combination of these could perhaps find other bugs that individual mutations could not. We ranked the mutations of the previous experiment first by their ability to find divergent behaviour, and then by their ability to produce crashing programs. We took the top 5 of these mutations—*VirtualRegisterChanger*, *RandomInstructionGenerator*, *InstructionSwapper*, *InstructionDuplicator*, and *BranchShifter*—and ran DEXFUZZ with all 5 enabled, each with equal chance of being selected. This time, we ran DEXFUZZ for 10,000 iterations, and for each divergence we found, we checked to see if a combination of mutations was responsible for the divergence.

Table 3 presents the breakdown of execution outcomes when running the top 5 most successful mutations. As would be expected, the largest percentage of programs produced were invalid, since three of the top five mutations had extremely high invalid production rates. Like those mutations, divergences were also found. For each divergence found, we assessed if the divergence was produced by a combination of mutations, but found that this was not the case - in each

Execution Outcome	Observed
Verification Failure	9249
Timeout	129
Mutation Failure	28
Crashing	314
Successful	257
Divergent	23
Iteration Total	10000

Table 3. Breakdown of execution outcomes when running DEXFUZZ with only the top 5 most successful mutations enabled: VirtualRegisterChanger, RandomInstructionGenerator, InstructionSwapper, InstructionDuplicator, and BranchShifter.

program, a single instance of mutation created the divergent behaviour. It is easy to imagine a set of mutations that may result in divergence together - for example, creating a random legal *array-length* instruction, and then swapping it to a location where it becomes illegal and finds the bug mentioned in Section 1.1, where the swap without the *array-length* instruction would have also been legal. However, we have failed to find such occurrences in practice thus far, and so further investigation is required. More sophisticated mutations, such as those mentioned in Section 6 or fuzzing the *optimizing* compiler may provide better results.

3.4 Comparison with other mutative fuzzing strategies

While we showed in Section 2 that a naive fuzzing scheme was not ideal for fuzzing a strictly verified bytecode format, and while we have found no available fuzzing systems that specifically fuzz DEX programs for ART or Dalvik, there do exist target-program-agnostic fuzzers that attempt to intelligently perform mutative fuzzing. One such fuzzer is American Fuzzy Lop (AFL)[1], a system that fuzzes inputs for an instrumented binary program, observes the execution paths of the instrumented binary, and attempts to intelligently fuzz new inputs using basic bit-level flipping, such that it will aim to fuzz interesting programs further to explore new execution paths. It aims to maximise the number of unique execution paths it finds, as well as the number of crashes.

We have built a host (x86-64) version of ART with the compiler wrapper that AFL provides, that inserts the instrumentation required to be used with the version of AFL. We have specifically disabled the header verification of ART for this experiment as well, in order to give AFL a chance to explore more than the checksum calculation code of ART, as this is a known limitation of AFL. We ran AFL in its instrumented fuzzing mode for 24 hours, with a subset of 16 ART test programs as seeds, to see how many crashes and hangs it found, as well as programs with unique paths. Of these programs with unique paths, we ran them through our differential tester to see if any of the programs produced divergence between the interpreter's and the *quick* compiler's output. We then ran DEXFUZZ for 24 hours with the same seed programs, to see how many crashes, hangs, and divergent programs it produced. In this experiment, DEXFUZZ's mutations have all been set to have an equal chance of being triggered.

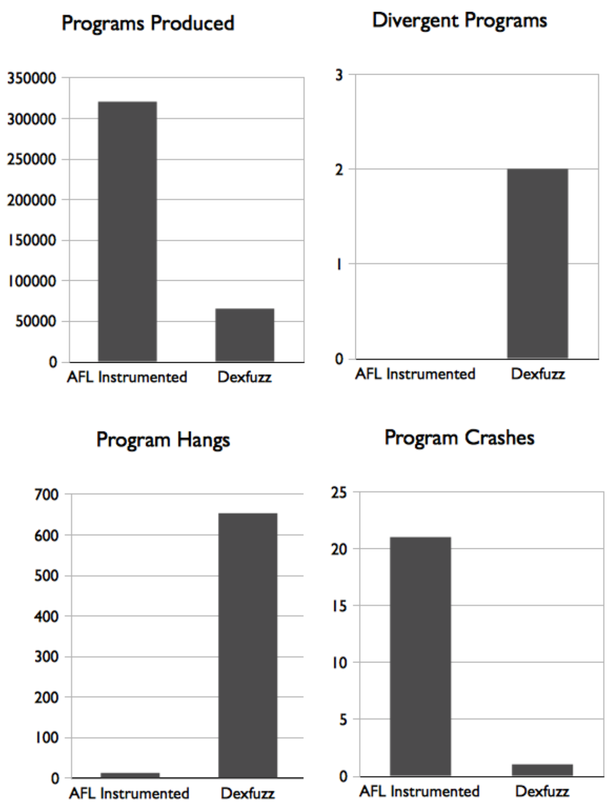

Figure 4. Comparison of success rates of AFL and DEXFUZZ after a 24-hour period of fuzzing.

Figure 4 shows how DEXFUZZ's ability to find divergent programs and crashes compares with AFL's. While it is clear that AFL is capable of producing a significantly larger number of programs within 24 hours, this could be explained by choice of implementation language (DEXFUZZ is written in Java, while AFL is written in C), and that very little time has currently been spent optimising DEXFUZZ, as most of the testing time in the tool is dominated by uploading programs to the test platform, and waiting for them to finish executing. Despite a comparatively small quantity of tests with respect to AFL, DEXFUZZ managed to find two programs that demonstrate divergent behaviour in ART while none of the programs that AFL marked as having unique execution paths demonstrated divergent behaviour. Indeed, of all the programs that AFL produced, none of them passed the verification stage of ART either. For all the hanging programs that AFL reported, none of these passed verification, and may have been an artefact of AFL's default timeout value. Despite having a more lenient timeout value, DEXFUZZ was able to find many more hanging programs in comparison to AFL, due to it actually producing programs that pass verification. Where AFL does excel in comparison to DEXFUZZ however, is its ability to find crashing behaviour. AFL has probably thoroughly tested the structural verification of ART, while DEXFUZZ's fuzzing is weaker in this regard. In general, however, we have shown that the use of domain-aware mutation is useful in discovering divergent behaviour.

4. Impact

In this paper, we have presented DEXFUZZ, a differential mutative fuzz testing tool for finding bugs in ART, and presented a number of bugs discovered in the AOSP version of ART. We have also presented some information about how many divergent programs we could have found in the experimental version of ART released in Android KitKat, if DEXFUZZ had been available at the time. Now we will discuss some of the impact on ART from our testing.

4.1 Improvements to ART Test Suite

Previously, the ART test suite consisted of a number of Java programs, that were compiled to DEX bytecode and executed. Our testing has uncovered bugs that are revealed using programs that cannot be produced directly from Java code, but require manipulation of already existing DEX bytecode. Since we started submitting patches to fix bugs uncovered by DEXFUZZ, the developers of ART in AOSP have added a new suite of tests to the existing ART suite, that allows for tests using mutated DEX bytecode to be checked. Now tests can be described using the *smali* DEX bytecode assembly format[6], which allows us to test for regressions in bugs found by DEXFUZZ. Our patches to AOSP were the catalyst for the creation of these tests, and we always aim to submit any patches with relevant tests for the test suite now.

4.2 Potential for future backends

Currently DEXFUZZ has only been put into practice with the *quick* compiler and interpreter of ART, as these are the only mature backends currently available. From looking at the AOSP code base, it is clear that significant work is going towards producing a new backend, the *optimizing* compiler. The *quick* backend treats DEX bytecode compilation in a simple manner, mapping directly from DEX bytecodes to sequences of native code and performing a few simple code optimizations on top. By comparison, the *optimizing* compiler focuses on much more significant optimization, of the DEX bytecode itself before lowering to native code, using much more sophisticated code optimization techniques, such as those found in modern JIT compilers. With such complexity comes the potential for bugs however, and as we have demonstrated DEXFUZZ's ability to find bugs in older versions of ART, we expect use of this tool to find bugs in the *optimizing* compiler early.

4.3 Contribution to AOSP

Since we have started developing DEXFUZZ, we have found a number of bugs in ART, and submitted patches for these bugs to the AOSP tree, most of which have been accepted, showing that DEXFUZZ finds bugs that the developers are interested in fixing. DEXFUZZ itself has been successfully submitted as a patch and merged into the AOSP code base, clearly indicating that DEXFUZZ is of considerable worth to the future development of ART.

5. Related Work

The use of fuzzing as a testing strategy started with Purdom[14], where a system that generates random sentences to test parsers was presented. Fuzzing was then used to test UNIX utilities for vulnerabilities by Miller et al.[13]. There have been a number of recent approaches to black-box fuzz testing for file formats, such as Peach[5] and AFL[1], as well as approaches to fuzzing of x86 instructions[11]. In this literature review we shall focus on the prior art of applying fuzzing to the testing of compilers and runtimes, considering other mutative approaches and the use of differential testing.

5.1 Differential testing of compilers

In [16], Yang et al. present Csmith, a tool that can generate millions of C99 compliant programs, and find over 300 bugs in open-source and commercial C compilers. They use differential testing to find these bugs, checking for a consensus in output between these different compilers, which inspired our use of multiple "backends" in the ART runtime. However, their main concern is with avoiding undefined behaviours of C programs, and as such their system does not actually use all features of any C standard. Because our tests are based on ART's available test suite, at a minimum we will test all the features of the VM that the test suite uses, which we then test in new combinations to explore for defects. We aim to quickly generate and throw away many programs that do not verify, in the hope of finding some that erroneously do verify, and lead to bugs. Finally, while their differential testing work focuses only on testing compilers, we test the verifier, compiler and complete runtime of a VM.

In [12], Vu et al. present Orion, a tool that uses Equivalence Modulo Inputs (EMI) to reveal more miscompilations than Csmith, which mainly highlights compiler crash defects. EMI is a form of mutative fuzzing, using code coverage information about programs to prune test programs into smaller programs with equivalent semantics. The set of equivalent programs are then executed with the same compiler to ensure they all produce the same result, with differences indicating compiler bugs. Using coverage information to direct mutations would be an interesting direction to take our work into in the future. Again, a concern of Orion is avoiding generating programs with undefined behaviours, a constraint we do not need to worry about.

5.2 Fuzz testing of runtimes

The published fuzzing of a Java virtual machine has been attempted before, in the tool jFuzz by Jayaraman et al. [10], built upon an explicit-state Java model checker and a system that fuzzes Java source code to find new program paths. However, we have not found any published literature concerning the application of fuzz testing at the bytecode-level for JVMs or the Dalvik/ART VMs. One advantage of fuzzing bytecode rather than source code is that features of the bytecode that the source language doesn't use can be tested. For example, while the *invoke_dynamic* JVM instruction isn't used by Java, it is used by other source languages that compile to JVM bytecode. Some preliminary research was performed into fuzzing ART by Sabanal at the Hack in the Box Amsterdam 2014 conference[4], but was not formally published.

Research has been made into other virtual machines that take source code rather than bytecode as their primary input. In particular, these tools have applied mutative and generative fuzzing to the source code using grammars. In [9], Holler et al. present LangFuzz, a language-agnostic system that has been tested against the JavaScript and PHP runtimes. The authors of [8] introduce the use of constraint logic programming into fuzzing, building upon the generation goals of stochastic context-free grammars that systems like LangFuzz use for program generation. They apply this technique to the JavaScript virtual machine. Our contribution in this paper, domain-aware binary fuzzing, is related to the use of stochastic context-free grammars, and it would be interesting to see if DEXFUZZ mutations could be described using context-free grammars in general.

The authors of [15] apply fuzz testing to the ActionScript virtual machine, and is probably most similar to our work, since ActionScript VMs accept compiled bytecode as input rather than source code. The authors' approach is to produce nearly valid ActionScript source code, parse this source and perform runtime class mutations to produce valid ActionScript programs, which then test the ActionScript VM. This approach is ultimately generative, and the authors compare its achieved code coverage against an existing test suite for ActionScript, whereas we use a test suite as a set of program seeds to perform our fuzzing.

6. Future Work

While DEXFUZZ has been submitted and merged into the ART code base in AOSP, we have a number of plans for extensions and improvements to DEXFUZZ.

As it is theoretically possible for bugs to arise from a set of multiple DEX mutations, we plan to add a feature to DEXFUZZ where once a divergent program is found, DEXFUZZ will automatically search for the minimal set of mutations that result in execution divergence. This can be done by, for example, finding if the mutations can be limited to a single method, and then performing a binary search from there. This will aid users of the tool in finding the source of the bug, if they know exactly which methods and mutations are responsible for the issue.

In the case of multi-threaded programs, it is possible that while a test suite seed program produces deterministic output during normal execution, mutation could cause the program to produce non-deterministic output, either during normal execution, or during the process of error reporting. Currently DEXFUZZ will filter out any found divergences between backends due to this problem, by checking if one of the backends produces divergent output with itself when tested multiple times with the same program. However, a better solution might be to run a backend multiple times to find the range of divergent outputs it produces, and check that the other backends' outputs fall within this set of divergent outputs, such as performed in QuickCheck[7].

DEXFUZZ's testing strategy is currently predicated on the belief there are no bugs in the VM such that all backends of the VM will produce exactly the same incorrect output. In this case, there would be no divergence to indicate the presence of a bug. This is a general flaw with the differential testing methodology, and the only possible mitigation is to use more backends to perform testing. Alternatively, another VM that obeys the same specification could be used. While this is easily possible when testing JVMs, with the myriad of implementations available, the only feasible execution alternative to ART on Android is the previous VM, Dalvik, although there exist alternative Dalvik implementations such as Myriad's Alien Dalvik[2] and BlueStacks AppPlayer[3]. It would be useful to execute programs with both the ART and Dalvik VMs, provided that the compared output is filtered to remove known divergences between the two implementations - namely the way that they report errors.

Finally, we plan to add more mutations to DEXFUZZ, including "global" mutations such as changing the access flags of classes, methods, and fields. For example, the volatile flag of fields could be flipped randomly, to ensure that the loading and storing to and from the fields always holds the correct semantics in presence of other optimizations, although again this may require measures to check for self-divergence. Additional method-level mutations are planned, such as injecting the console printing of virtual register values at random locations, or calls to specific system methods such as the `System.gc()` method, to exercise random calls to the garbage collector.

7. Conclusion

In this paper we have presented a tool to augment the testing of the new ART VM in Android, called DEXFUZZ. We have shown how use of this tool in the past could have prevented the inclusion of security bugs in the original experimental release of ART. We have also described a number of bugs that we have found and fixed using this tool in the AOSP version of ART, to demonstrate the utility of this tool to prevent bugs being released in future versions of the software. DEXFUZZ is built upon the combination of domain-aware binary fuzzing and differential testing, and we have evaluated what sorts of binary mutations that we have applied to DEX files lead to divergent behaviour in the experimental version of ART found in Android KitKat, to confirm which mutations were more useful for finding bugs. We find that mutating use of virtual registers, insertion of random instructions, and swapping instructions, while increasing the amount of invalid programs produced, are the mutations most likely to lead to divergent behaviour, and should be useful initial mutations to use if future VM designers wish to use this testing approach.

References

[1] american-fuzzy-lop on Google Code. `https://code.google.com/p/american-fuzzy-lop/`. Accessed: 2014-11-12.

[2] Alien Dalvik. `http://www.myriadgroup.com/products/device-solutions/mobile-software/alien-dalvik/`. Accessed: 2014-11-25.

[3] BlueStacks AppPlayer. `http://www.bluestacks.com/app-player.html`. Accessed: 2014-11-25.

[4] HITB2014AMS - Day 1 - State of the ART: Exploring the new Android KitKat Runtime. `https://www.corelan.be/index.php/2014/05/29/hitb2014ams-day-1-state-of-the-art-exploring-the-new-android-kitkat-runtime/`. Accessed: 2014-11-20.

[5] Peachfuzzer. `http://peachfuzzer.com/`. Accessed: 2014-11-20.

[6] smali - an assembler/disassembler for android's dex format on Google Code. `https://code.google.com/p/smali/`. Accessed: 2014-11-19.

[7] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. . URL `http://doi.acm.org/10.1145/351240.351266`.

[8] K. Dewey, J. Roesch, and B. Hardekopf. Language fuzzing using constraint logic programming. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 725–730, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8. . URL `http://doi.acm.org/10.1145/2642937.2642963`.

[9] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2362793.2362831`.

[10] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jfuzz: A concolic whitebox fuzzer for java. In *NASA Formal Methods*, pages 121–125, 2009.

[11] A. Lanzi, L. Martignoni, M. Monga, and R. Paleari. A smart fuzzer for x86 executables. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, SESS '07, pages 7–, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2952-6. . URL `http://dx.doi.org/10.1109/SESS.2007.1`.

[12] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 216–226, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. . URL `http://doi.acm.org/10.1145/2594291.2594334`.

[13] 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. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/96267.96279`.

[14] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972. ISSN 0006-3835. . URL `http://dx.doi.org/10.1007/BF01932308`.

[15] G. Wen, Y. Zhang, Q. Liu, and D. Yang. Fuzzing the actionscript virtual machine. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 457–468, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1767-2. . URL `http://doi.acm.org/10.1145/2484313.2484372`.

[16] 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*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. . URL `http://doi.acm.org/10.1145/1993498.1993532`.