TOWARDS FULLY AUTOMATED `object`-VERIFICATION

IOhannes m zmölnig
Institute of Electronic Music and Acoustics
University of Music and Dramatic Arts
Graz, Austria
zmoelnig@iem.at

ABSTRACT

While other sound synthesis systems have made the switch to 64-bit floating point values for signal processing a while ago, Pure Data (Pd) seems to be stuck with single precision numbers. An initial port to 64-bit precision has been presented in 2011, but 5 years later little progress has been made. One of the alleged showstoppers for switching to 64-bit precision is the plethora of available 3rd party externals, many of which might start malfunctioning in subtle ways. In this work we try to solve this problem by means of automatic `object` verification.

Keywords. unit tests, fuzz testing, automatic testing, double precision Pd, Pure data

1. INTRODUCTION

In 2006, Csound5 was released and featured a `Csound64` flavor, which uses `double` precision signal processing throughout [6]. It took until 2011, when Vetter created an initial 64-bit precision build of Pd [7], but 5 years later we are practically still using only single precision builds, with the `double` precision amendments that made it into the Pd-core silently bit rotting away.

There are a number of reasons why we see so little adoption. One of the alleged showstoppers for switching to 64-bit precision is the plethora of available 3rd party externals, many of which might appear to work fine (e.g. it is possible to compile these external with setting the Pd-data types to `double` precision). But when those objects are actually used, they might turn out to be fundamentally broken [8].

The two obvious ways to deal with this problem are to either optimistically introduce the change and hope that people will report any problems so they can promptly be fixed (at the expense that the users will have a broken system until all those issues are fixed), or to proactively inspect the source code of all those externals for potential problems with the proposed change. Neither of the two ways is especially appealing.

In software industries the „standard" way to automatically verify whether a piece of code is working as expected (e.g. after doing a major refactoring of the source code), is by running it through a number of *unit tests*, that check whether a „program" produces the expected result when fed with known input data.

Unit tests for Pd are nothing new: a few libraries come with their own ad-hoc unit test suites (e.g. `zexy`, including a bash-script for running about 100 unit tests since late 2005 [11] (although about 70% of the tests check for trivial parameters like existence), or `PuREST JSON`, featuring a python-based system since 2014 with 20 functional tests).

There have also been several proposals for formalized unit test frameworks, starting with `PureUnity` [1] up to `testtools` [3].

Incidentally, the latter was designed (among other things) to provide a means to automatically verify whether an object is fit for `double` precision Pd.

One big disadvantage of unit testing is that somebody has to write the actual unit tests, which is time consuming and unrewarding, resulting in poor adoption: of more than 160 libraries found on the Pure Data SVN on Sourceforge, only six (6!) have a test suite than can be run automatically.

However, poor adaption of unit testing can also be observed in the software industry in general [4]. This has led to the development of automatic testing systems, like `American Fuzzy Lop` [9], which automatically creates a corpus of test data that will test as many lines of code as possible. These systems are normally used to find security flaws and crasher bugs, by feeding unexpected input data to a program. Notably, such unsupervised testing is not targeted at verifying whether a program produces the *correct* output given a known input. However, a synthesized test corpus can be used to verify, whether the software being tested behaves the same in different *environments*.

The underlying assumption is, that if we have a test corpus that - when fed into a software unit running in an environment A - triggers execution of all possible code paths in that unit, then the unit's behavior is fully determined. If we then use this very same test corpus on that unit in a different environment B and the unit's response is the „same" (within limits), then the object can be considered to work the **same** in both environments. Finally, if the unit is already known to work **correctly** in environment A, we can conclude that it is then also working **correctly** in environment *B*.

For Pd, those *units* could be single objects, while *environments* could be different supported platforms (Linux, OSX, W32,…), but they could also be different flavors of Pd, such as the traditional single precision Pd or the `double` precision Pd. Furthermore, such *environments* could be the different forks of the Pd-engine, such as Pd-devel, Desire Data, Pd-extended, Pd-l2ork or Purr Data. Given that many Pd objects have been heavily tested over the last few years on Pd-vanilla (and Pd-extended), we assume that they are working **correctly** (though not necessarily bug-free) in this environment.

2. GOAL

This project's goal is the automatic creation of test corpora for any Pd object. We are targeting primarily at externals (compiled objects, written in C), since the *environments* we want to investigate (single resp. `double` precision Pd) are mainly differing on the low, binary level. If all objects that interact with the low-level data representation are working correctly, we don't expect any environment-specific problems with objects that only operate on the high level.

However, the methods we are using can equally be applied to abstractions.

3. DESIGN

For the task at hand we created the `PeDAnT` framework (an acronym for *Pure Data Automatic Testing* framework).

The purpose of this testing framework is to generate a corpus of test data, and to provide a way to run each test case through the object: e.g. by sending messages or signals to its `inlet`s, or letting time pass. It also needs to provide a way to record all output of the object (whatever message or signal was sent to its `outlet`s at which time), so test runs in different environments can be compared.

3.1. Corpus Synthesis

The most complicated part in this endeavor is certainly the creation of a test corpus, that covers all aspects of our object. For this task, we propose to utilize fuzz testing tools. In *fuzz testing*, software is tested by feeding it generated random data as input, and watching the software's reaction. It is mostly used to detect vulnerabilities, such as memory leaks or even crashes that depend on the input data. However, naïvely generating random data is unlikely to find many of these „interesting" edge cases in finite time. A promising technique to solve this problem is to use program-flow analysis of the tested binary as a fitness indicator for a genetic algorithm, which is used to (per)mutate a small starting corpus.

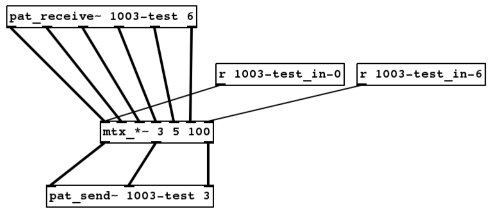

Figure 1. Automatically connecting all iolets of `mtx_*~ 3 5 100` under test

One popular (and fast) tool that utilizes this approach is Michał Zalewski's *American Fuzzy Lop* (`afl-fuzz`), which uses code instrumentation to generate a test corpus that triggers the execution of a maximum number of source code lines in the tested program. `afl-fuzz` also features a „fork server" to rapidly start numerous test runs (e.g. we found that it could start up Pd up to 1000 times per second - with Pd opening a test patch, loading the test data, feeding data into the test object and the quitting).

However, despite the astonishing performance of the test runs and a corpus generation that is sped up by code coverage analysis, fuzz testing is still a very laborious undertaking: testing a single program can easily take days!

Since `afl-fuzz` will keep generating data forever, a termination condition is needed. Using `gcov` (the test coverage program that comes with the *GNU Compiler Collection*), it is easy to create a (human readable) report on how many lines of code are already covered by the test corpus. Once a maximum of code is covered, the test corpus can be considered complete. In practice this maximum will often be below 100%, mostly because some branches in the code are simply unreachable (e.g. a debugging function that is compiled but not actually used).

3.2. `PeDAnT` Framework

A small support library has been written that instantiates an object, fully connects its iolets (Fig.1) and then feeds test data to the object and records any response it gets.

This library is used in an abstraction `pedant-run.pd` that executes a single test run and then quits Pd. It is to be used in `-batch` mode for speedy execution (this is important as the tests also include (randomized) time, so running a test in *real time* might take years).

The rest consists of a set of Python scripts, mainly wrapping a correct invocation of `afl-fuzz` (and `gcov`), so it uses the Pd-patch to instantiate the tested object and feed it the synthesized test corpus.

There are also scripts to feed a (completed) corpus to a tested object and record it's responses, and to compare the responses of two test runs (e.g. within different environments).

Most scripts take a configuration file (Listing 1) that can be shared among the various tasks.

```
[pedant]
subject=limiter~
[pd]
#binary=/usr/bin/puredata
args=-path /home/pedant/src/zexy/src/
[afl-fuzz]
tests=workbench/limiter~/seedtest/
#dict=workbench/limiter~/dict/
out=workbench/limiter~
[afl-cov]
binary=/home/pedant/bin/afl-cov
codedir=/home/pedant/src/zexy/src/
args=--coverage-at-exit
```

Listing 1: `PeDAnT` shared configuration

3.3. Test Input Data

So far we found three different types of input data that determine the behavior of an object. Any concrete object may not necessarily require all three types to be fully determined.

3.3.1. Signals:

Signals consist of a series of numbers (signal blocks). For every DSP-tick all `inlet~`s need to be filled) with full signal blocks (each of equal length).

3.3.2. Messages:

Messages consist only of serializable data (numbers and symbols), that are send immediately („now") to the specified `inlet` of the object.

3.3.3. Time:

The *time* data type allows to schedule *messages* (and *signals*) by advancing the logical time („now"). Some objects exhibit their specific behavior only over time (e.g. `delay`), and this data type provides a way to make (logical) time pass.

3.4. File Format

`afl-fuzz` is capable of synthesizing test data of almost arbitrary complexity [10]. However, the synthesizing algorithm mostly alters existing test cases by randomly flipping bits, or by cutting up test cases and splicing them together anew.

A file format with the following characteristics should greatly improve the speed of test data generation:

- expressive: randomly flipping bits should create (very) *different* test-cases.
- robust: randomly splicing test-data should create *valid* test-cases again.
- precise: it should be easy to express any number and any string.
- extensible: it should be easy to add new data types.

These requirements led to the design of a binary format as described in Listing 2.

To allow for an arbitrary number of data points (e.g. messages or signals) each holding arbitrary data (including typical `EOL` characters like `CRLF` or `NUL`), each data point is SLIP-encoded [5]. The data representation of the actual values are modeled closely after the values found in current (32-bit) Pd-vanilla: all *numbers* (including samples) are 32-bit floating point values. Timestamps are stored as `double` precision (64-bit) floating point values, and are actually time increments (in [ms]): in order to guarantee a monotonic clock those values must therefore always be positive.

Signals are stored as an arbitrary length list of numbers. The actually used signal samples are then calculated by repeating the sequence as needed, until all `inlet~`s are filled. In order to fully determine the signal input to an object, an additional parameter *signalblocksize* is therefore required. To keep the signal block size in a reasonable range, the actual value stored is an exponent in the range 0..16 (which gives possible block sizes 1..65536).

Data generated by an object during a test run can be stored for later comparison by prefixing each data frame with a `*`. When reading a data file *as input*, any lines prefixed with `*` are simply ignored. (In general, unknown prefixes are ignored).

4. WORKFLOW

4.1. Generating Input Data

The central task of the framework is the synthesis of a meaningful test corpus. This is done with the help of the fuzzer tool `afl-fuzz`.

4.1.1. Instrumenting the Tested Objects

`afl-fuzz` works best if it can instrument the binary under test, by adding *marker points* that allow a quick evaluation whether a given code path has been executed or not. Instrumenting the binary also allows `afl-fuzz` to apply its *fork server* during the tests, which greatly speeds up testing as it doesn't need to spawn a new process for each test run.

```
; data points are SLIP-encoded before being stored in the file
datapoint = direction (message / timeincrement / signal / signalblocksize)
direction = ["*"] ; the optional '*' prefix indicates result data (not used as input)
message = %s"m" iolet *(string / float)
timeincrement = %s"t" float64 ; positive delta time in [ms]
signal = %s"s" *float32
signalblocksize = %s"b" bsize ; the actual bufsize is 2^bsize
string = %s"s" *CHAR *1%x00
float = %s"f" float32
float32 = 4OCTET ; 4byte float (big-endian)
float64 = 8OCTET ; 8byte float (big-endian)
iolet = OCTET ; unsigned char
bsize = OCTET ; unsigned char
```

Listing 2: ABNF of the test file format

Instrumenting is done by by building the object with a special compiler `afl-gcc` [1] , an enhanced variant of `gcc`. Any reasonable build system allows to override the compiler via the `CC` (or `CXX`, in case of C++-projects) variable.

4.1.2. Seeding the Test Data

In order to quickly create „interesting" test data, the genetic algorithm needs to be seeded with some initial „meaningful" test cases. For this purpose we pursued the following strategies, all of which can at least be semi-automated:

1. Harvesting existing (help-)patches for messages that make sense to the object: For this we simply extracted *any* message (as found in `message boxes`) from patches that use the object, and created one test case per found message. We relied on the splicing capabilities of the fuzzer to create meaningful message sequences from these solitary messages.

2. Keyword dictionaries: `afl-fuzz` can use dictionaries of keywords to build additional tests. By extracting strings/symbols from the source code and binary files of the object, we built a small set of keywords that are hopefully useful when testing the object.

3. Existing tests: Many Pd objects have similar interfaces. Therefore the test corpus of one object can serve as a good starting point for another object. However, this can quickly give a too large seed corpus, which will slow down the genetic algorithm, especially if many tests do **not** fit the object's interface (e.g. a message for which the object has no method). `afl` provides a tool for *corpus minimization*, which will remove superfluous tests from a corpus that do not trigger any new execution paths.

[1] or `afl-clang` if that is preferred, though we haven't used that in our tests so far.

4.1.3. Synthesizing Data

Once a suitable seed corpus has been created, the fuzzer can start generating a full test corpus.

```
pedant-fuzz \
  --config mytest.conf \
  --fuzz-out <inputcorpus>
```

The fuzzer will then start to generate thousands of test cases, and run them through the tested object. Whenever it detects that new execution paths have been triggered by a given test, it will use that test as the new seed for generating even more tests.

Targeting the fuzzer at the `zexy`'s `limiter~`, it took about 10 days (with 4 synchronized fuzzer instances) to generate a total of 11643 test cases (see Fig.2).

4.1.4. Minimizing Input Data

The generated corpus is usually rather large and covers a lot of redundant tests. Many items in the corpus will contain garbage data (due to the random nature of the test generation), which can easily be eliminated by parsing the test file, discarding invalid data points and saving the remaining ones. Any duplicates created during this process can be safely eliminated.

Also, `afl` provides a corpus minimization tool, `afl-cmin`, which can further reduce the corpus to a minimum set of files that covers all code paths.

```
pedant-cmin \
  --config mytest.conf \
  <inputcorpus> <reducedcorpus>
```

This was able to reduce the 11643 test cases for `limiter~` to 696. (while at the same time reducing the memory footprint from 90MB to 3.4MB).

4.1.5. Creating Tests

Once a sufficiently large corpus of test data has been generated, one more test run is needed to generate

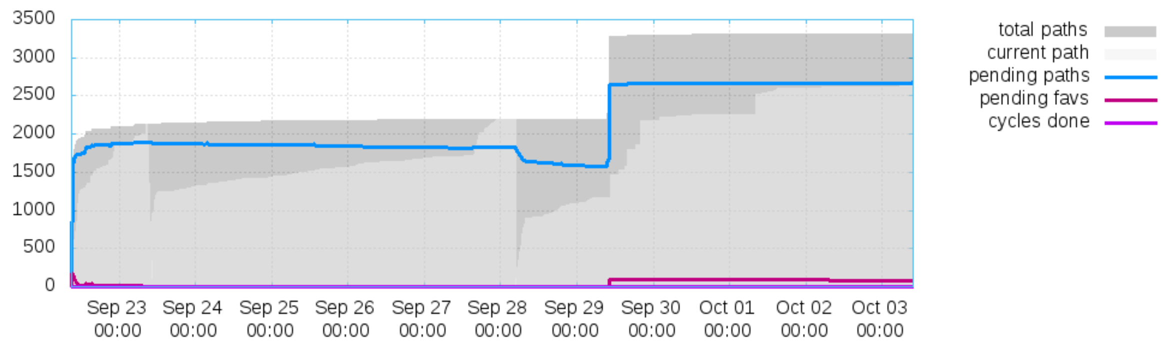

Figure 2. code paths discovered by one of four parallel fuzzer instances over the period of 10 days, targeting **zexy**'s `limiter~`. After one week, a fundamentally new code path was discovered, creating a new set of test-cases to be explored.

a set of *comparable* tests. This time, the output of the tested `object` is stored alongside its input, thus recording the entire interaction of the object (modulo side effects). Output data has the very same format and available types as input data (see 3.3) (though this time the `outlet` number is stored along with a message), but is marked with an asterisk (`*`) prefix within the file.

```
pedant-run \
  --config mytest.conf \
  <inputcorpus> <resultsA>
```

4.2. Running Tests

The test corpus (consisting of input and output data) can then be used to verify the object in a different environment, simply by using it as input (discarding any *output* data present in the corpus) and storing the results in a different directory:

```
pedant-run \
  --config mytest.conf \
  <resultsA> <resultsB>
```

4.3. Comparing Results

Verifying the `object` is done by comparing `<resultsA>` and `<resultsB>`, data item by data item: if any of the *output* data differs the test has failed (if the *input* data differs, the test run itself was faulty). Symbols and integer values in the test data, are compared for absolute equality, but floating point data needs special consideration [2] and is compared with both an *absolute* and a *relative* (in relation to the reference (input) data) ϵ. One can also specify a value for ∞ (any data that has a bigger absolute value, will be treated as inf). This is useful, since an intermediate inf value can poison the final result, and when doing `double` precision calculations this poisoning is less likely to happen.

```
pedant-compare \
  --absolute-tolerance 3e-5 \
  --relative-tolerance 1e-5 \
  --infinity 1e30 \
  <resultsA> <resultsB>
```

5. DISCUSSION

5.1. Interpreting Failed Tests

Even when fuzzily comparing test results (as discussed in 4.3), verifying a `double` precision build of `limiter~` against a `single` precision reference still yields a few false positives. These happen with very large input values that make the `single` precision binary unstable. There are a number of false positives like this, where a failing test is actually desired behavior, as it hints at the advantages of the tested environment: e.g. that it is possible to do things with a `double` precision environment that are impossible to do in a `single` precision environment.

It turns out that interpreting the result of a failed test (the differences between the output data sets generated by multiple test runs) is surprisingly hard, as it lacks meaningful context data. Is the failed test a false positive? If it hints at a real problem, where to start looking for in the source code?

5.2. Performance

So far, we have only conducted experimental test runs on a very small scale. The hardware used to conduct the tests was an Intel®Core™i7-870 CPU, running at 2.93GHz, where 4 cores where used to run four fuzzer instances in parallel.

The corpus synthesis for a slightly complex object like `limiter~` already took quite a lot of time - in the order of days to weeks, raising the question on how useful the proposed technique would be in testing the hundreds of objects commonly used.

On the other hand, simple objects like `sgn~` take considerably less time: starting from a single generic seed test case, it took less than five minutes to create 200 test cases (before minimization) that cover 97.2% of the entire code.

5.3. Automation

As of now, the automation of the framework consists of a few scripts that still require some human intervention. Most notably the corpus generation requires supervision to determine the halting condition.

5.4. Limitations of Testable Objects

Currently the only kind of objects that can be tested without user intervention are *functional* objects. That is, only objects that generate their output solely from their input data (and their internal state, as defined by creation arguments). This excludes any object whose acquire input data from external sources (such as files, user-input, hardware,...). Also, the tests depend on the object sending any output through its `outlet`s and currently do not check for any side-effects (such as created files, or controlled hardware).

Objects that depend on other objects (e.g. `r foo` or `tabread bar`) *can* be tested, but require a (human) agent to create a functional *abstraction* that wraps the interaction of the object with it's peers and presents an interface where only serializable data is sent via `inlet`/`outlet`s.

5.5. The Bad News: Code Coverage as a Terminating Condition

Unfortunately, it was easy to proof that the primary assumption of the framework (that by executing a maximum of code-paths we can reliably trigger all bugs that produce different results on single resp. `double` precision builds) can be falsified.

E.g. running aggressive minimization on the corpus for `limiter~`, reduced the number of test cases to 260, which cover **98.1%** of the `limiter~.c` source code (a local maximum: the remaining 7 lines of code were never executed because they were generally unreachable or because the creation arguments to `limiter~` were kept fixed). Running those tests in `single` resp. `double` precision environments revealed 11 test cases that would produce different results, thus failing the tests.

Removing these interesting test cases from the set (thus producing a non-failing test corpus) and reevaluating the covered code-paths resulted in a coverage of **98.1%**, which is exactly the same coverage as when those test cases were included.

Therefore, maximizing the code-coverage is not a sufficient strategy to catch all errors introduced by `double` precision builds (since the generated test corpus could have well not-included the interesting test cases, and still would have been considered „complete”). This also suggests that aggressive corpus minimization based on coverage analysis, might eventually drop interesting test cases.

5.6. The Good News

While the initial goal of verifying objects across environments cannot be achieved reliably, the proposed methodology is still able to generate interesting test corpora that can help improve the implementation of the tested object.

For instance, running the fuzzer on a couple of `zexy` objects quickly revealed a systematic programming error that would trigger a crash of Pd. Consequently, these have now been fixed.

Despite not having a reliable metric for the corpus completeness regarding different precisions, we found that in practice the large number of generated tests makes it still *probable* that at least one of the tests will expose issues in the problem domain.

Also, for different problem domains (e.g. comparing different implementations of the Pd-engine, like Pd-vanilla vs Pd-l2ork) the initial assumption might still hold true.

6. FUTURE WORKS

As shown in 5.5, code coverage is not a sufficient metric to determine whether a test corpus will expose certain problems. More research is required to find an indicator that can reliably guide the test synthesis algorithm to produce test cases that expose problems with 64-bit precision builds.

6.1. Harvesting for Seed Data

The current approach to generate an initial seed corpus for the fuzzer uses human-guided brute force. It includes all messages the happen to occur in some help patch, regardless of whether these messages are actually sent to the object under test. Better data might be harvested by actually running such a patch, triggering all user-interaction (e.g. by randomly clicking any message-boxes, toggles and similar) and recording all messages as they are received by the tested object (e.g. by replacing the test-subject with a dummy object with identical interfaces).

Also the compiled object itself could be harvested more efficiently than just extracting all the strings (which returns a lot of garbage). A good start would be to check which message selectors are accepted on which inlet (and what argument they take).

6.2. Testing Creation Arguments

So far the creation arguments of an object are fixed, and test synthesis does not vary them during the test runs. Since some objects have parameters that can only be set via creation arguments (e.g. the number of iolets), this possibly leaves a substantial part of the code untested.

We are currently investigating the most stable way to allow the fuzzer to also control the way an object is instantiated. One choice is to interpret a message at the very beginning of the test data as creation arguments.

6.3. Intelligent Corpus Minimization

The current strategy for corpus minimization (code coverage analysis) has been shown to be problematic. A simple strategy to remove garbage from the test data is partly applied, by reading a data file, discarding any unknown messages and writing it back to disk. This can further be improved, by discarding any message that is not understood by the tested object (The `no method for 'bong'` case).

This will also create more meaningful seed corpora for new objects from existing test sets.

6.4. Interpretation Help

In most cases, one has to understand *why* a test fails in order to fix the code. However, the current representation of failed test results (basically the differing values are displayed in a human readable format, value by value) is not very helpful. A representation with diff-highlighting that also provides a bit of context depending on the data type, will most likely make the interpretation of the test results easier. For signal data, a graphical representation might make even more sense.

6.5. A Central Repository for Test Corpora

On the long run it would be nice to have a central repository of test corpora for „all" objects. This would make it possible to quickly validate new environments. Having a largish set of tests available, can also speed up the synthesis of a new corpus for a yet-untested object.

7. CONCLUSIONS

We have presented the `afl`-based fuzz-test framework `PeDAnT` for semi-automated testing of Pd-objects. The main strength of the framework is that little human interaction is required for the chores of *writing tests*: `PeDAnT` is able to synthesize test corpora with a maximum code coverage of the tested object on its own, trying to find as many different issues as possible.

While our original objective - fully verifying an object's functionality for `double` precision builds of Pd - has not (yet) been met, the framework can give useful hints when problems are likely. Even without comparing test results from different environments, the test framework has already proven useful for finding a number of crasher bugs.

`https://git.iem.at/pd/pedant`

8. REFERENCES

[1] M. Bouchard. PureUnity [online]. 2005. URL: `https://git.puredata.info/cgit/svn2git/libraries/pureunity.git/` [accessed 2016-10-05].

[2] B. Dawson. Comparing Floating Point Numbers, 2012 Edition [online]. 2012. URL: `https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/` [accessed 2016-10-05].

[3] F. J. Kraan and K. Vetter. testtools [online]. 2011. URL: `https://puredata.info/downloads/testtools` [accessed 2016-10-05].

[4] X. Qu and B. Robinson. A Case Study of Concolic Testing Tools and their Limitations. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 117–126. IEEE, 2011.

[5] J. L. Romkey. A Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP. RFC 1055, RFC Editor, June 1988. URL: `http://www.rfc-editor.org/rfc/rfc1055.txt`.

[6] B. Vercoe, J. ffitch, J. Piché, P. Nix, R. Boulanger, R. Ekman, D. Boothe, K. Conder, S. Yi, M. Gogins, and A. Cabrera. *The canonical Csound reference manual.* MIT Media Lab, 2007.

[7] K. Vetter. Double precision Pd [online]. 2011. URL: `http://www.katjaas.nl/doubleprecision/doubleprecision.html` [accessed 2016-10-05].

[8] J. Wilkes. What's the deal with [utime] object? [online]. 2016. URL: `https://lists.puredata.info/pipermail/pd-list/2016-02/113637.html` [accessed 2016-10-05].

[9] M. Zalewski. American Fuzz Lop [online]. 2013. URL: `http://lcamtuf.coredump.cx/afl/` [accessed 2016-10-05].

[10] M. Zalewski. Finding Bugs in SQLite, the Easy Way [online]. 2015. URL: `http://lcamtuf.blogspot.co.at/2015/04/finding-bugs-in-sqlite-easy-way.html` [accessed 2016-10-05].

[11] IO. m. zmölnig. zexy [online]. 2005. URL: `https://git.iem.at/pd/zexy` [accessed 2016-10-05].