Opus Testing

Opus Testing

- Goal:
  - Create a high quality specification *and* implementation
- Problem:**Engineering is hard**
  - More details than can fit in one person's brain at once
  - Does the spec say what was meant?
  - Does what was meant have unforeseen consequences?
  - Are we legislating bugs or precluding useful optimizations?

Why we need more than formal listening tests

- Formal listening tests are expensive, meaning
  - Reduced coverage
  - Infrequent repetition
- Insensitivity
  - Even a severe bug may only rarely be audible
  - Can’t detect matched encoder/decoder errors
  - Can’t detect underspecified behavior (e.g., “works on my architecture”)
  - Can’t find precluded optimizations

The spec is software

- The formal specification is 29,833 lines of C code
  - Use standard software reliability tools to test it
- We have fewer tools to test the draft text
  - The most important is reading by multiple critical eyes
  - This applies to the software, too
  - Multiple authors means we review each other's code

Continuous Integration

- The later an issue is found
  - The longer it takes to isolate the problem
  - The more risk there is of making intermediate development decisions using faulty information
- We ran automated tests continuously

All | **opus** | +

S	Name ↓	Description	Last Statuses	Last Duration	
	opus	Libopus autotools build Rendered library documentation: HTML, PDF	**6.4 days** > 6.4 days	3 min 19 sec	
	opus build from IETF site	This fetches the draft posted on the IETF site and builds libopus from it	9.9 days	21 sec	
	opus-arm	Libopus autotools build on ARM	13 days	13 min	
	opus-coverage	Libopus floating-point coverage test Latest results	**6.4 days** > 6.4 days	12 min	
	opus-coverage-fixed	Libopus fixed-point coverage test Latest results	6.4 days	13 min	
	opus-cppcheck	Libopus static analysis with cppcheck Latest results	6.4 days	26 min	
	opus-custom-fixed	Libopus opus-custom fixed-point autotools build	**6.4 days** > 6.4 days	2 min 56 sec	
	opus-custom-fixed-solaris	Libopus fixed-point autotools build on Solaris	4 days	57 min	

Software Reliability Toolbox

- No one technique finds all issues
- All techniques give diminishing returns with additional use
- So we used a bit of everything
  - Operational testing
  - Objective quality testing
  - Unit testing (including exhaustive component tests)
  - Static analysis
  - Manual instrumentation
  - Automatic instrumentation
  - Line and branch coverage analysis
  - White- and blackbox “fuzz” testing
  - Multiplatform testing
  - Implementation interoperability testing

Force Multipliers

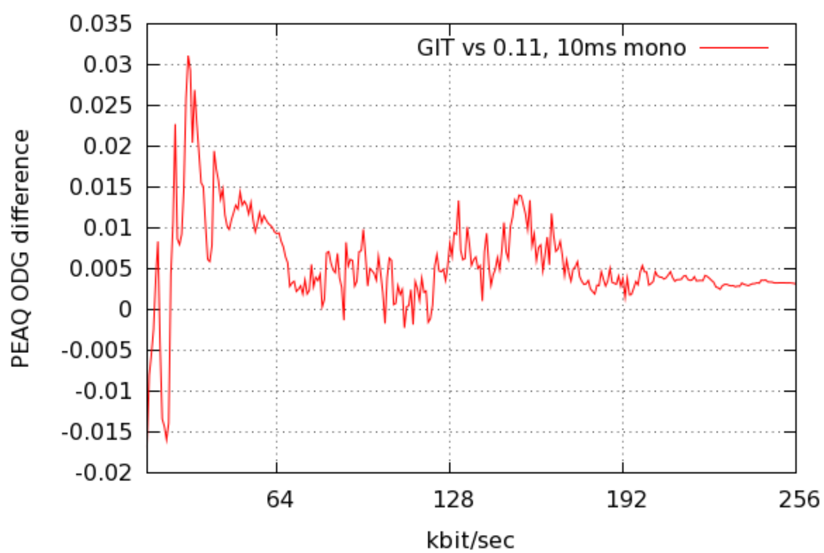
- All these tools are improved by more participants
  - Inclusive development process has produced more review, more testing, and better variety
  - Automated tests improve with more CPU
    - We used a dedicated 160-core cluster for large-scale tests
- Range coder mismatch
  - The range coder has 32 bits of state which must match between the encoder and decoder
  - Provides a “checksum” of all encoding and decoding decisions
  - *Very* sensitive to many classes of errors
  - opus_demo bitstreams include the range value with every packet and test for mismatches

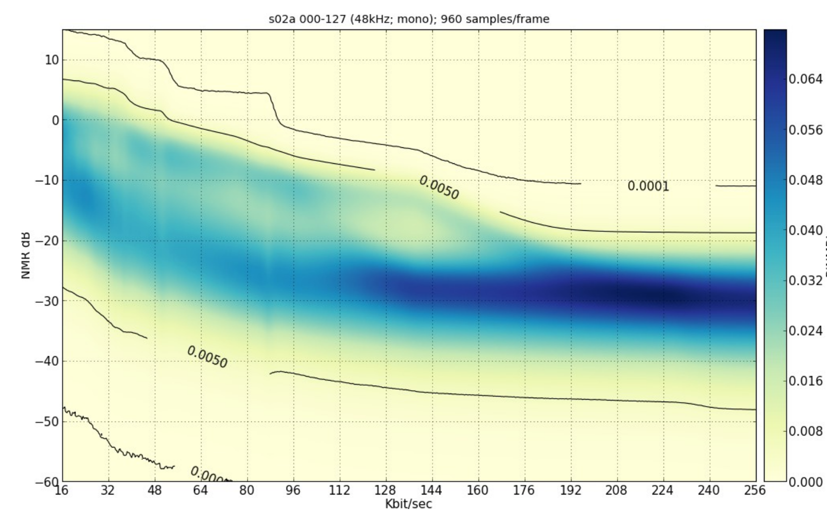
Operational Testing

- Actually use the WIP codec in real applications
- Strength: Finds the issues with the most real-world impact
- Weakness: Low sensitivity
- Examples:
  - “It sounds good except when there’s just bass” (rewrote the VQ search)
  - “It sounds bad on this file” (improved the transient detector)
  - “Too many consecutive losses sound bad” (made PLC decay more quickly)
  - “If I pass in NaNs things blow up” (fixed the VQ search to not blow up on NaNs)

Objective Quality Testing

- Run thousands of hours of audio through the codec with many settings
  - Can run the codec 6400x real time
  - 7 days of computation is 122 years of audio
- Collect objective metrics like SNR, PEAQ, PESQ, etc.
- Look for surprising results
- Strengths: Tests the whole system, automatable, enables fast comparisons
- Weakness: Hard to tell what's "surprising"
- Examples: See slides from IETF-80





Unit Tests

- Many tests included in distribution
  - Run at build time via “make check”
  - On every platform we build on
- Exhaustive testing
  - Some core functions have a small input space (e.g., 32 bits)
  - Just test them all
- Random testing
  - When the input space is too large, test a different random subset every time
  - Report the random seed for reproducibility if an actual problem is found
- Synthetic signal testing
  - Used simple synthetic signal generators to produce “interesting” audio to feed the encoder
  - Just a couple lines of code: no large test files to ship around
- API testing
  - We test the entire user accessible API
  - Over 110 million calls into libopus per “make check”
- Strengths: Tests many platforms, automatic once written
- Weaknesses: Takes effort to write and maintain, vulnerable to oversight

Static Analysis

- Compiler warnings
  - A limited form of static analysis
  - We looked at gcc, clang, and MSVC warnings regularly (and others intermittently)
- Real static analysis
  - cppcheck, clang, PC-lint/splint
- Strengths: Finds bugs which are difficult to detect in operation, automatable
- Weaknesses: False positives, narrow class of detected problems

Manual Instrumentation

- Identify invariants which are assumed to be true, and check them explicitly in the code
- Only enabled in debug builds
- 513 tests in the reference code
  - Approximately 1 per 60 LOC
- Run against hundreds of years of audio, in hundreds of configurations
- Strengths: Tests complicated conditions, automatic once written
- Weaknesses: Takes effort to write and maintain, vulnerable to oversight

Automatic Instrumentation

- valgrind
  - An emulator that tracks uninitialized memory at the bit level
  - Detects invalid memory reads and writes, and conditional jumps based on uninitialized values
  - 10x slowdown (600x realtime)
- clang-IOC
  - Set of patches to clang/llvm to instrument all arithmetic on signed integers
  - Detects overflows and other undefined operations
  - Also 10x slowdown
- All fixed-point arithmetic in the reference code uses macros
  - Can replace them at compile time with versions that check for overflow or underflow
- Strengths: Little work to maintain, automatable
- Weaknesses: Limited class of errors detected, slow

Line and Branch Coverage Analysis

- Ensures other tests cover the whole codebase
- Logic check in and of itself
  - Forces us to ask why a particular line isn't running
- We use condition/decision as our branch metric
  - Was every way of reaching this outcome tested?
- "make check" gives 97% line coverage, 91% condition coverage
- Manual runs can get this to 98%/95%
  - Remaining cases mostly generalizations in the encoder which can't be removed without decreasing code readability
- Strengths: Detects untested conditions, oversights, bad assumptions
- Weaknesses: Not sensitive to missing code

```
      [ +  + ]:   15462414 :        if (N>1)
             :             :        {
             :    12684510 :           excess = IMAX(bits[j]-cap[j],0);
             :    12684510 :           bits[j] -= excess;
             :             :
             :             :           /* Compensate for the extra DoF in stereo */
[ +  + ][ +  + ]: 12684510 :           den=(C*N+ ((C==2 && N>2 && !*dual_stereo && j<*intensity) ? 1 : 0));
[ +  + ][ +  + ]
```

Decoder Fuzzing

- Blackbox: Decode 100% random data, see what happens
  - Discovers faulty assumptions
  - Tests error paths and “invalid” bitstream handling
  - Not very complete: some conditions highly improbable
  - Can’t check quality of output (GIGO)
- Partial fuzzing: Take real bitstreams and corrupt them randomly
  - Tests deeper than blackbox fuzzing
- We’ve tested on hundreds of years worth of bitstreams
- Every “make check” tests several minutes of freshly random data
- Strengths: Detects oversights, bad assumptions, automatable, combines well with manual and automatic instrumentation
  - Fuzzing increases coverage, and instrumentation increases sensitivity
- Weaknesses: Only detects cases that blow up (manual instrumentation helps), range check of limited use
  - No encoder state to match against for a random or corrupt bitstream
  - We still make sure different decoder instances agree with each other

Whitebox Fuzzing

- KLEE symbolic virtual machine
  - Combines branch coverage analysis and a constraint solver
  - Generates new fuzzed inputs that cover more of the code
- Used during test vector generation
  - Fuzzed an encoder with various modifications
  - Used a machine search of millions of random sequences to get the greatest possible coverage with the least amount of test data
- Strengths: Better coverage than other fuzzing
- Weaknesses: Slow

Encoder Fuzzing

- Randomize encoder decisions
- More complete testing even than partial fuzzing (though it sound bad)
- Strengths: Same as decoder fuzzing
  - Fuzzing increases coverage, and instrumentation increases sensitivity
- Weaknesses: Only detects cases that blow up (manual instrumentation helps)
  - But the range check still works

Multiplatform Testing

- Tests compatibility
- Some bugs are more visible on some systems
- Lots of configurations
  - Float, fixed, built from the draft, from autotools, etc.
  - Test them all
- Automatic tests on
  - Linux {gcc and clang} x {x86, x86-64, and ARM}
  - OpenBSD (x86)
  - Solaris (sparc)
  - Valgrind, clang-static, clang-IOC, cppcheck, lcov
- Automated tests limited by the difficulty of setting up the automation
  - We had 28 builds that ran on each commit

Additional Testing

- Win32 (gcc, MSVC, LCC-win32, OpenWatcom)
- DOS (OpenWatcom)
- Many gcc versions
  - Including development versions
  - Also g++
- tinycc
- OS X (gcc and clang)
- Linux (MIPS and PPC with gcc, IA64 with Intel compiler)
- NetBSD (x86)
- FreeBSD (x86)
- IBM S/390
- Microvax

Toolchain Bugs

- All this testing found bugs in our development tools as well as Opus
  - Filed four bugs against pre-release versions of gcc
  - Found one bug in Intel's compiler
  - Found one bug in tinycc (fixed in latest version)
  - Found two glibc (libm) performance bugs on x86-64

Implementation Interop Testing

- Writing separate decoder implementation
- Couldn’t really finish until the draft was “done”
- CELT decoder complete
  - Implements all the MDCT modes
  - Floating-point only
  - Shares no code with the reference implementation
  - Intentionally written to do things differently from the reference implementation
  - Bugs during development used to tune opus_compare thresholds
  - Also revealed several “matched errors” in the reference code
  - Currently passes opus_compare on the one MDCT-only test vector
  - Tested with over 100 years of additional audio
    - 100% range coder state agreement with the reference
    - Decoded 16-bit audio differs from reference by no more than ±1

Implementation Interop Testing

- SILK decoder in progress
  - Started last Thursday
  - Implemented from the draft text (not the reference implementation)
  - Code is complete
  - Range check passes for bitstreams tested so far (not many)
  - Actual audio output completely untested
- Hybrid modes: coming soon