# Kcov - a single-step code coverage tool

Simon Kågström

Net Insight
https://github.com/SimonKagstrom/kcov

September 20, 2018

# Motivation

```
    if ((err = ReadyHash(&SSLHashMD5, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashMD5.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashMD5.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashMD5.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    if ((err = SSLHashMD5.final(&hashCtx, &hashOut)) != 0)
        goto fail;
}
else {
    /* DSA, ECDSA - just use the SHA1 hash */
    dataToSign = &hashes[SSL_MD5_DIGEST_LEN];
    dataToSignLen = SSL_SHA1_DIGEST_LEN;
}

hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;

if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(ctx,
                   ctx->peerPubKey,
                   dataToSign,                /* plaintext */
                   dataToSignLen,             /* plaintext length */
                   signature,
                   signatureLen);
if(err) {
    sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                "returned %d\n", (int)err);
    goto fail;
}

fail:
```

# Motivation

```
 96
 97    0 / 1          if ((err = ReadyHash(&SSLHashMD5, &hashCtx)) != 0)
 98    0 / 1              goto fail;
 99    0 / 1          if ((err = SSLHashMD5.update(&hashCtx, &clientRandom)) != 0)
100    0 / 1              goto fail;
101    0 / 1          if ((err = SSLHashMD5.update(&hashCtx, &serverRandom)) != 0)
102    0 / 1              goto fail;
103    0 / 1          if ((err = SSLHashMD5.update(&hashCtx, &signedParams)) != 0)
104    0 / 1              goto fail;
105    0 / 1          if ((err = SSLHashMD5.final(&hashCtx, &hashOut)) != 0)
106    0 / 1              goto fail;
107              }
108              else {
109                  /* DSA, ECDSA - just use the SHA1 hash */
110    1 / 1          dataToSign = &hashes[SSL_MD5_DIGEST_LEN];
111    1 / 1          dataToSignLen = SSL_SHA1_DIGEST_LEN;
112              }
113
114    1 / 1      hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
115    1 / 1      hashOut.length = SSL_SHA1_DIGEST_LEN;
116    1 / 1      if ((err = SSLFreeBuffer(&hashCtx)) != 0)
117    0 / 1          goto fail;
118
119    1 / 1      if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
120    0 / 1          goto fail;
121    1 / 1      if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
122    0 / 1          goto fail;
123    1 / 1      if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
124    0 / 1          goto fail;
125    1 / 1      if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
126                  goto fail;
127    1 / 1          goto fail;
128              if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
129                  goto fail;
130
131              err = sslRawVerify(ctx,
132                                 ctx->peerPubKey,
133                                 dataToSign,                /* plaintext */
134                                 dataToSignLen,            /* plaintext length */
135                                 signature,
136                                 signatureLen);
137              if(err) {
138                  sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
139                              "returned %d\n", (int)err);
140                  goto fail;
141              }
142
143          fail:
```

# More motivation

- Programming is hard!
- To improve software quality, good tooling and methodology is important

# Outline

# Table of Contents

# Code coverage terminology

- **Function coverage**: Functions in the program
- **Statement coverage**: Statements in the program
- **Branch coverage**: All ways through branches (if/case)
- **Condition coverage**: All boolean expressions evaluated to both true/false
- **Modified condition/decision coverage (MC/DC)**: Each decision takes every possible outcome

(From Wikipedia)

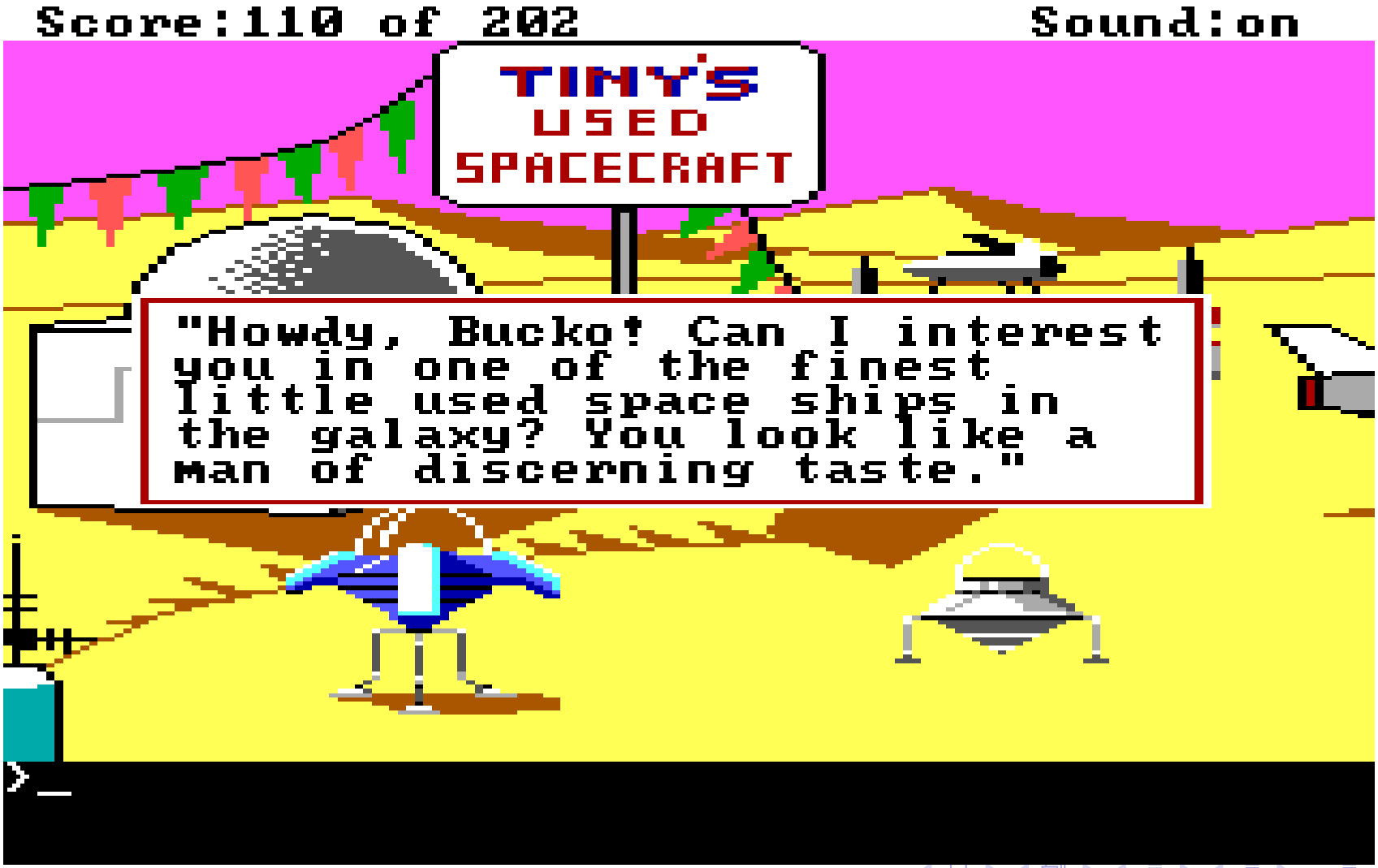
# Problems with traditional tools

- gcov + lcov is a multi-step process
- gcov leaves droppings after compilation/running
- A program which crashes will not generate coverage data

## Example

```
$ gcc -g -Wall --coverage goto-fail.c
$ ./a.out
$ ls
  a.out  goto-fail.c  goto-fail.gcda  goto-fail.gcno
$ lcov --capture --directory project-dir --output-file coverage.info
$ genhtml coverage.info --output-directory out
```

# Instead...


Score:110 of 202
Sound:on
TINY'S
USED
SPACECRAFT
"Howdy, Bucko! Can I interest you in one of the finest little used space ships in the galaxy? You look like a man of discerning taste."
>_

# Table of Contents

# Kcov overview

- Kcov started as a fork of Bcov by Thomas Neumann in 2010
- Bcov doesn't rely on compile-time instrumentation, but instead uses debug information
- Bcov was a great idea, but I thought it could be improved upon

# Kcov overview

- Kcov started as a fork of Bcov by Thomas Neumann in 2010
- Bcov doesn't rely on compile-time instrumentation, but instead uses debug information
- Bcov was a great idea, but I thought it could be improved upon
- Can you guess why it's called kcov?

# Kcov overview

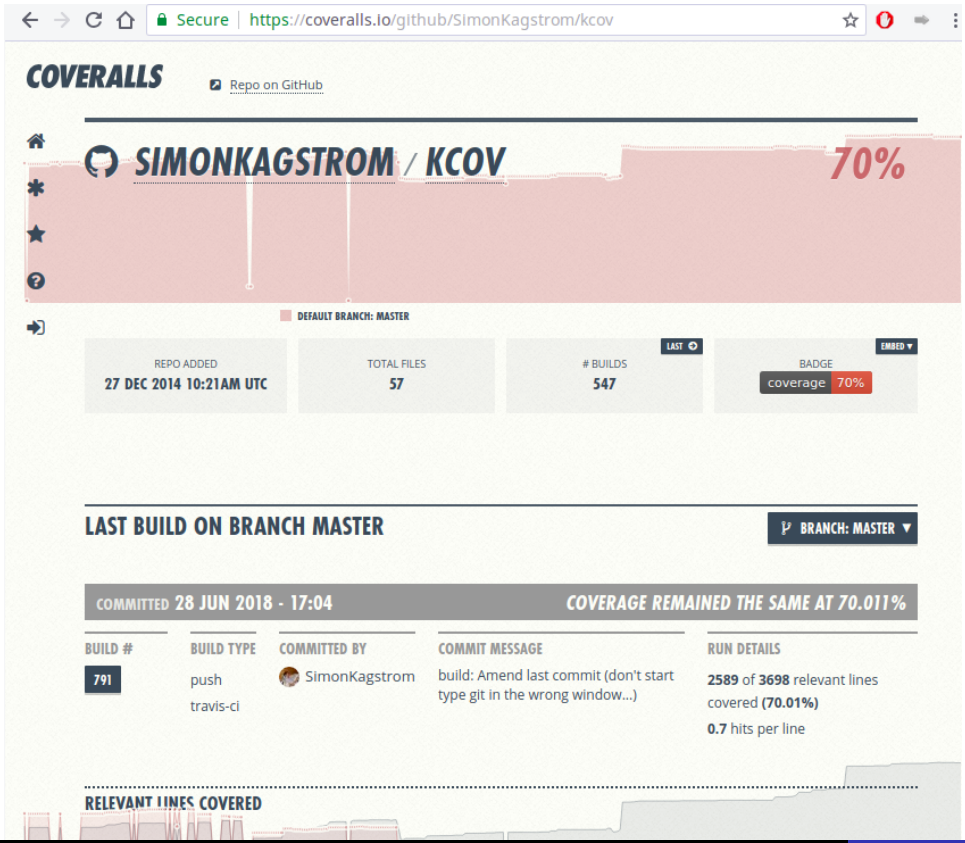
- Kcov started as a fork of Bcov by Thomas Neumann in 2010
- Bcov doesn't rely on compile-time instrumentation, but instead uses debug information
- Bcov was a great idea, but I thought it could be improved upon
- Can you guess why it's called kcov?
- It is not related to the kernel Kcov (and predates it by many years)

# Main features of kcov

- Supports collection and reporting of binaries as long as debug information is present
- Automatically collects coverage from shared libraries
  - Both linked into the binary, and opened via **dlopen**
- Collection and reporting is done in a single step
- Generates HTML output as well as several XML formats for integration in other environments
- Accumulates coverage information between runs
- Automatically merges multiple binaries into a combined report
  - Merging of multiple reports can also be done by the tool
- Works on Linux, FreeBSD and Mac OSX on x86, ARM and PowerPC

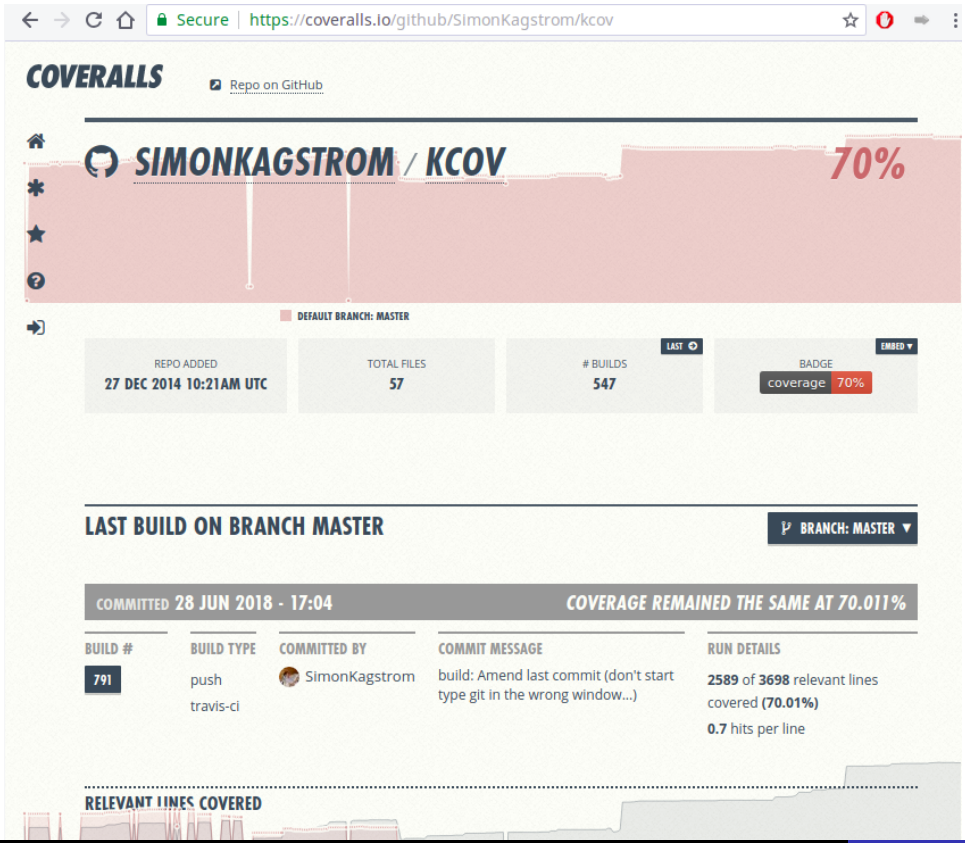
# Integration with CI systems

- Jenkins and SonarQube output is generated by kcov
- Uploading to Coveralls.io is built-in
- Uploading to Codecov.io is supported by the upstream project



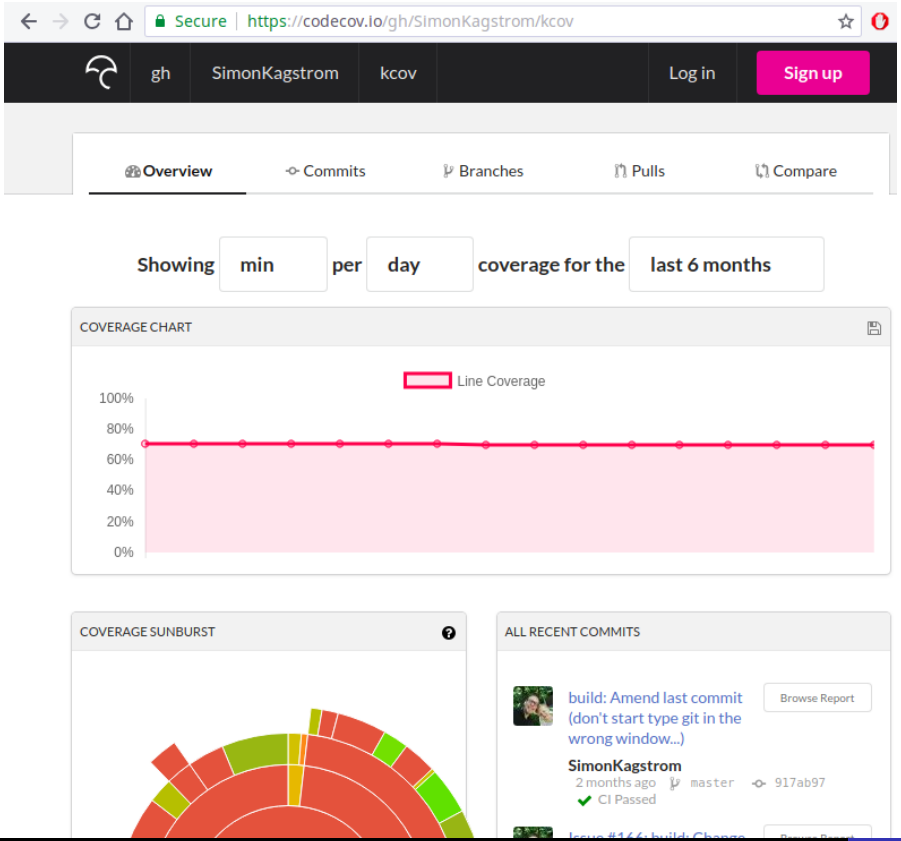
# Integration with CI systems

- Jenkins and SonarQube output is generated by kcov
- Uploading to Coveralls.io is built-in
- Uploading to Codecov.io is supported by the upstream project



# Integration with CI systems

- Jenkins and SonarQube output is generated by kcov
- Uploading to Coveralls.io is built-in
- Uploading to Codecov.io is supported by the upstream project



# Python and Bash code coverage

- Kcov can also collect coverage for Python and Bash
- For Bash and Python, line counts are reported

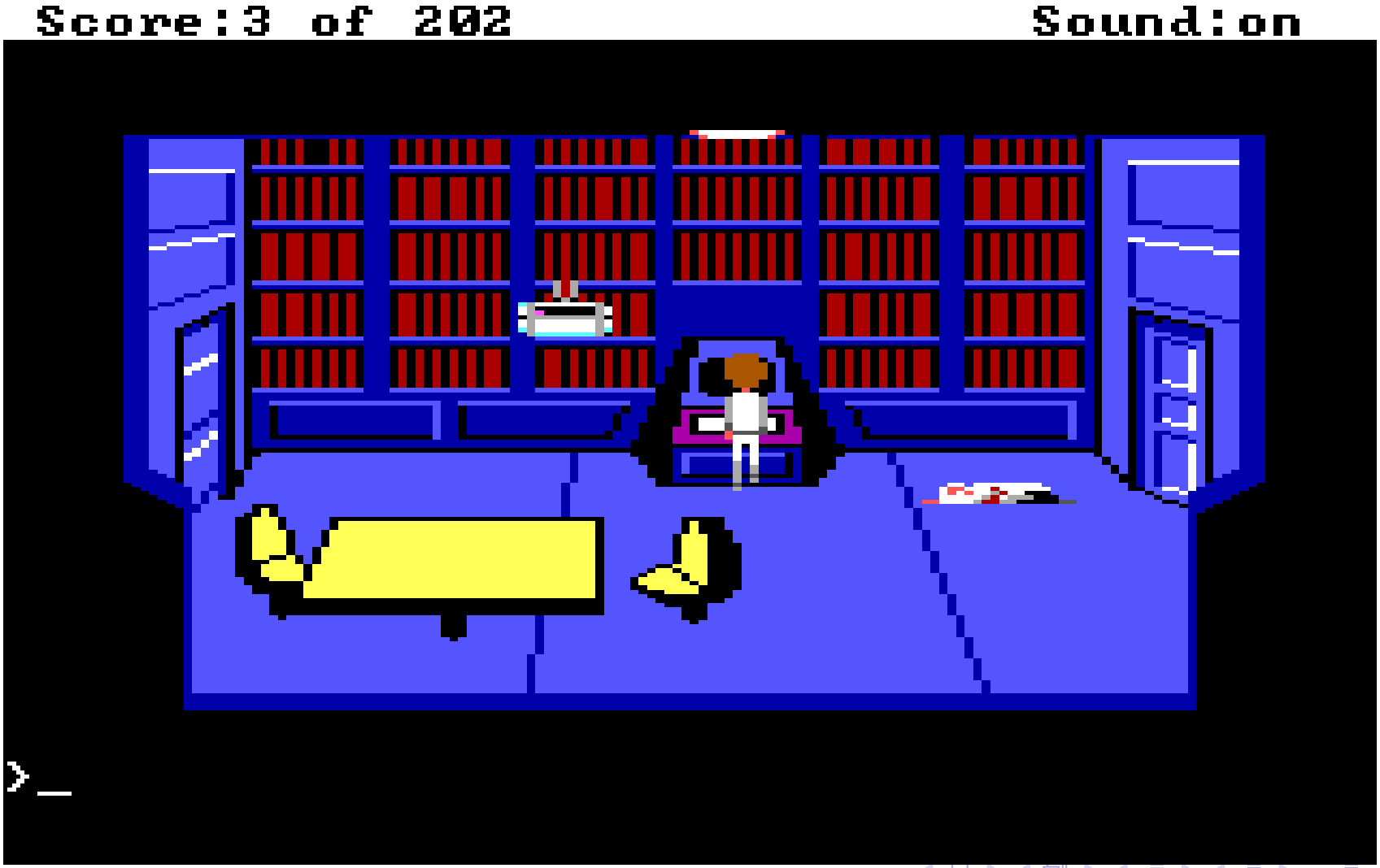
**Command:** other.sh
**Date:** 2018-09-17 20:59:01
**Code covered:** 85.0%
**Instrumented lines:** 20
**Executed lines:** 17

```
1          #!/bin/sh
2
3       1  echo "This is another shell script"
4
5       1  if [ $1 -eq 5 ] ; then
6       0    echo "Kalle anka"
7          fi
8
9       0  for var in "$@" ; do
10      0    echo $var
11         done
12
13      1  round=0
14     22  for (( ; ; )) do
15     11    round=$(($round + 1))
16     11    if [ $round -gt 10 ] ; then
17      1            break
18           fi
19     10    echo "hej $round"
20         done
21
22         echo "Not covered"  # LCOV_EXCL_LINE
23      1  echo "Covered"
24
25         # Ranges
26         echo "Not covered"  # LCOV_EXCL_START
27         echo "Also not covered"
28         echo "Also not covered?" # LCOV_EXCL_STOP
29      1  echo "Covered"
30
```

**Command:** testdriver
**Date:** 2018-09-17 21:00:06
**Code covered:** 88.9%

```
1          #!/usr/bin/python
2
3       2  import sys
4       1  import unittest
5
6       3  class testcase (unittest.TestCase):
7
8       2      def setUp(self):
9       1          print('in setUp()')
10      1          self.unitundertest = unitundertest.testme()
11
12      2      def testMethod1(self):
13      1          print('in testcase.testMethod1()')
14      1          expected=4
15      1          result= self.unitundertest.testmethod1(2,2)
16
17      1          self.assertEqual(expected,result)
18
19
20      1  if __name__ == '__main__':
21      1      sys.path.append('.')
22
23      1      try:
24      1          import unitundertest
25      0      except:
26      0          print('exception')
27
28      1      unittest.main()
```

# Interactive Demo!



# Table of Contents

# Dwarf stabs Mach-O Elf

- **ELF** is a binary format used on many Unices
  - Both for object files, shared libraries and executable files
  - Contains the code, data, constants and relocation information
- **Mach-O** is the binary format used on Mac OS X
- **Dwarf** is a format for debug information
  - Contains the mapping between source lines and addresses
  - Type information for variables etc

# Instrumenting binaries with kcov

- DWARF contains file:line to address records, which kcov uses to set breakpoints

## Example

```
 124:     return (void*)op_create(RPAR);
400ff7:       bf 30 00 00 00          mov    $0x30,%edi
400ffc:       e8 1f 05 00 00          callq  401520 <op_create>
401001:       e9 20 ff ff ff          jmpq   400f26 <ts_next_token+0x46>
401006:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
40100d:       00 00 00
 125: p_state->last_token_is_od = 0;
401010:       c7 43 08 00 00 00 00    movl   $0x0,0x8(%rbx)
 126: return (void*)op_create(op);
401017:       bf 04 00 00 00          mov    $0x4,%edi
40101c:       e8 ff 04 00 00          callq  401520 <op_create>
```

# Instrumenting binaries with kcov

- DWARF contains file:line to address records, which kcov uses to set breakpoints

## Example

```
 124:     return (void*)op_create(RPAR);
400ff7:       bf 30 00 00 00          mov    $0x30,%edi
400ffc:       e8 1f 05 00 00          callq  401520 <op_create>
401001:       e9 20 ff ff ff          jmpq   400f26 <ts_next_token+0x46>
401006:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
40100d:       00 00 00
 125: p_state->last_token_is_od = 0;
401010:       c7 43 08 00 00 00 00    movl   $0x0,0x8(%rbx)
 126: return (void*)op_create(op);
401017:       bf 04 00 00 00          mov    $0x4,%edi
40101c:       e8 ff 04 00 00          callq  401520 <op_create>
```

# Instrumenting binaries with kcov

- DWARF contains file:line to address records, which kcov uses to set breakpoints
- So how are breakpoints set on Linux?

## Example

```
 124:     return (void*)op_create(RPAR);
400ff7:       bf 30 00 00 00          mov    $0x30,%edi
400ffc:       e8 1f 05 00 00          callq  401520 <op_create>
401001:       e9 20 ff ff ff          jmpq   400f26 <ts_next_token+0x46>
401006:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
40100d:       00 00 00
 125: p_state->last_token_is_od = 0;
401010:       c7 43 08 00 00 00 00    movl   $0x0,0x8(%rbx)
 126: return (void*)op_create(op);
401017:       bf 04 00 00 00          mov    $0x4,%edi
40101c:       e8 ff 04 00 00          callq  401520 <op_create>
```

# How does kcov know where to set the breakpoint, really?

- The ELF binary contains program code in sections, located at address found in the binary
- But shared libraries and PIEs complicate this

## Example

```
  $ readelf --wide -S ~/local/bin/calc
There are 36 section headers, starting at offset 0x7ec8:

Section Headers:
  [Nr] Name              Type            Address          Off    Size
  [...]
  [10] .rela.plt         RELA            0000000000400a88 000a88 000150
  [11] .init             PROGBITS        0000000000400bd8 000bd8 000017
  [12] .plt              PROGBITS        0000000000400bf0 000bf0 0000f0
  [13] .text             PROGBITS        0000000000400ce0 000ce0 001b42
  [14] .fini             PROGBITS        0000000000402824 002824 000009
  [15] .rodata           PROGBITS        0000000000402830 002830 0006f5
  [...]
```

# kcov on Linux, FreeBSD and Mac OSX

- ptrace is a truly archaic interface, and pretty non-portable
- libelf is also interesting: **elf_version** must be called at program start!
- Mac OSX has been implemented using the LLDB debugger as a library

## Example

```
static long getRegs(pid_t pid, void *addr, void *regs, size_t len)
{
#if defined(__aarch64__)
    struct iovec iov = {regs, len};
    return ptrace(PTRACE_GETREGSET, pid, (void *)NT_PRSTATUS, &iov);
#else
    return ptrace((__ptrace_request ) PTRACE_GETREGS, pid, NULL, regs);
#endif
}
```

# So are there any interesting bugs?

# Dwarf quirks

- Dwarf generation on Linux is sometimes buggy, containing invalid entries

## Example

```
 124:     return (void*)op_create(RPAR);
400ff7:       bf 30 00 00 00          mov    $0x30,%edi
400ffc:       e8 1f 05 00 00          callq  401520 <op_create>
401001:       e9 20 ff ff ff          jmpq   400f26 <ts_next_token+0x46>
401006:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
40100d:       00 00 00
 125: p_state->last_token_is_od = 0;
401010:       c7 43 08 00 00 00 00    movl   $0x0,0x8(%rbx)
 126: return (void*)op_create(op);
401017:       bf 04 00 00 00          mov    $0x4,%edi
40101c:       e8 ff 04 00 00          callq  401520 <op_create>
```

# Dwarf quirks

- Dwarf generation on Linux is sometimes buggy, containing invalid entries

## Example

```
 124:     return (void*)op_create(RPAR);
400ff7:       bf 30 00 00 00          mov    $0x30,%edi
400ffc:       e8 1f 05 00 00          callq  401520 <op_create>
401001:       e9 20 ff ff ff          jmpq   400f26 <ts_next_token+0x46>
401006:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
40100d:       00 00 00
 125: p_state->last_token_is_od = 0;
401010:       c7 43 08 00 00 00 00    movl   $0x0,0x8(%rbx)
 126: return (void*)op_create(op);
401017:       bf 04 00 00 00          mov    $0x4,%edi
40101c:       e8 ff 04 00 00          callq  401520 <op_create>
```

# LLDB and fork

- The OSX port has another interesting quirk: forks are unsafe
- LLDB doesn't catch breakpoints from children, and will therefore crash in the child after a fork

## Example

```
auto child = fork();
if (child < 0)
      printf("Error\n");
else if (child == 0)
      printf("In child\n"); // Breakpoint here is bad news!
else
{
      printf("In parent, child %d\n", child);
      wait(&status);
}
```

# LLDB and fork

- The OSX port has another interesting quirk: forks are unsafe
- LLDB doesn't catch breakpoints from children, and will therefore crash in the child after a fork
- How can we workaround that?

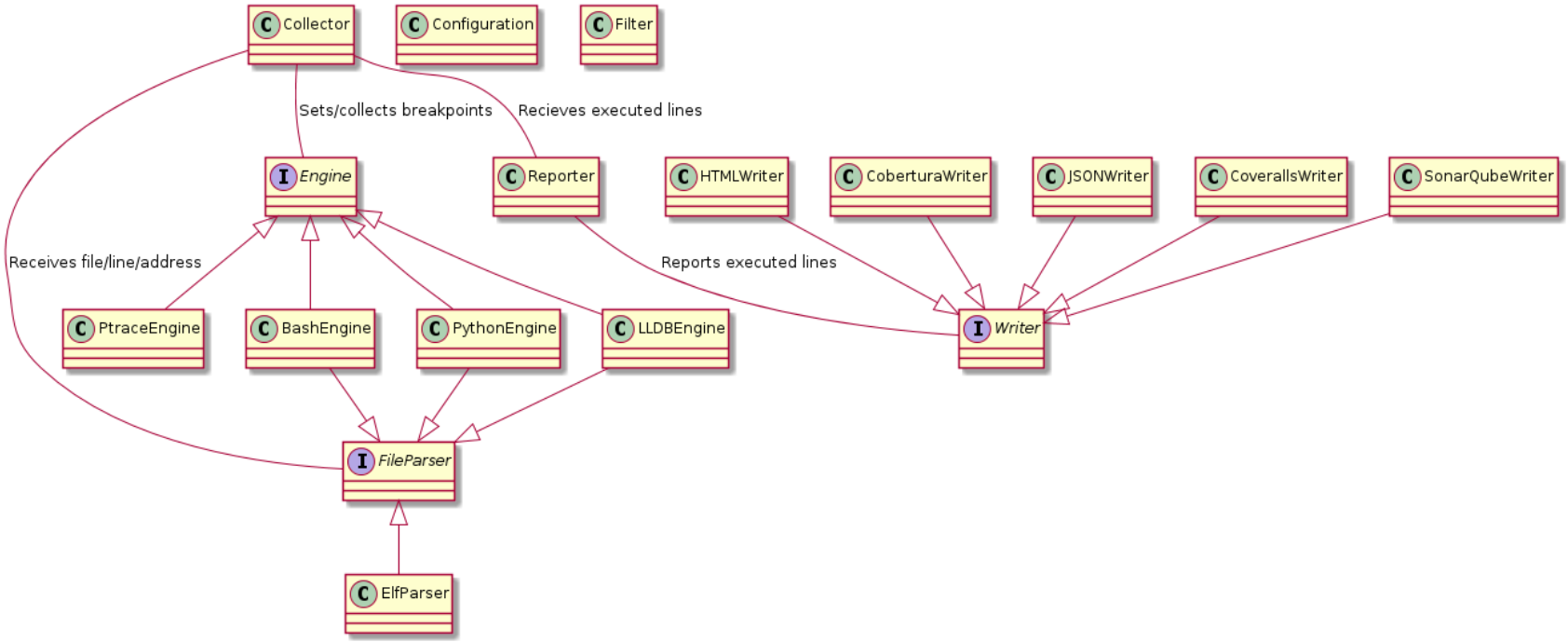
## Example

```
auto child = fork();
if (child < 0)
      printf("Error\n");
else if (child == 0)
      printf("In child\n"); // Breakpoint here is bad news!
else
{
      printf("In parent, child %d\n", child);
      wait(&status);
}
```

# So how does Python and Bash work?

- Bash has a **PS4** environment variable, which can be used to trace execution
- Python has a trace function which can be controlled via **sys.settrace**
- kcov needs to know what lines are "executable" and not, also in this setting
  - Easy in Python
  - *Very* tricky in Bash. Heredocs etc

# Design


Collector
Configuration
Filter
Sets/collects breakpoints
Recieves executed lines
Engine
Reporter
HTMLWriter
CoberturaWriter
JSONWriter
CoverallsWriter
SonarQubeWriter
Receives file/line/address
Reports executed lines
PtraceEngine
BashEngine
PythonEngine
LLDBEngine
Writer
FileParser
ElfParser

# Questions and comments!

(Images from http://www.falselogic.net/LetsPlay/SpaceQuest.html)