My name is Simon Kågström and I work at Net Insight as a developer of embedded software. Tonight I will present kcov, a code coverage collection and presentation tool for UNIX systems.

So let's get started!

Kcov - a single-step code coverage tool

Simon Kågström

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

September 20, 2018

I'll start with some motivation for the talk. Code coverage, as described by Wikipedia, is the degree to which the source code of a program is executed when a particular test suite executes, measured in percentage. Code coverage tools can typically show which source code lines which have been executed, and sometimes how many times they are executed.

So let's say you're the developer of a particular piece of code, as shown on the slide. You run some tests on it, and collect code coverage to see how well your test suite actually works. NEXT.

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:
```

I'll start with some motivation for the talk. Code coverage, as described by Wikipedia, is the degree to which the source code of a program is executed when a particular test suite executes, measured in percentage. Code coverage tools can typically show which source code lines which have been executed, and sometimes how many times they are executed.

So let's say you're the developer of a particular piece of code, as shown on the slide. You run some tests on it, and collect code coverage to see how well your test suite actually works. NEXT. After code coverage collection, you will get a report which looks something like this. Red lines denote executable lines which have not been executed, green lines are executed lines. White lines are non-executable lines.

Hmm... Hang on now, why is everything between line 128 and 143 marked as non-executable? Some of you might already have identified the code in question. This is the SSL/TLS bug known as "goto fail". Looking at line 127, you can see that the control flow will unconditionally jump to the fail label, so lines 128 to 143 are indeed unreachable.

Stands out pretty clear from the coverage report don't you think? Modern compilers will also issue a warning on code like this.

Motivation

```
    0 / 1           if ((err = ReadyHash(&SSLHashMD5, &hashCtx)) != 0)
    0 / 1               goto fail;
    0 / 1           if ((err = SSLHashMD5.update(&hashCtx, &clientRandom)) != 0)
    0 / 1               goto fail;
    0 / 1           if ((err = SSLHashMD5.update(&hashCtx, &serverRandom)) != 0)
    0 / 1               goto fail;
    0 / 1           if ((err = SSLHashMD5.update(&hashCtx, &signedParams)) != 0)
    0 / 1               goto fail;
    0 / 1           if ((err = SSLHashMD5.final(&hashCtx, &hashOut)) != 0)
    0 / 1               goto fail;
                    }
                    else {
                        /* DSA, ECDSA - just use the SHA1 hash */
    1 / 1           dataToSign = &hashes[SSL_MD5_DIGEST_LEN];
    1 / 1           dataToSignLen = SSL_SHA1_DIGEST_LEN;
                    }

    1 / 1       hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
    1 / 1       hashOut.length = SSL_SHA1_DIGEST_LEN;
    1 / 1       if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    0 / 1           goto fail;

    1 / 1       if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    0 / 1           goto fail;
    1 / 1       if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    0 / 1           goto fail;
    1 / 1       if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    0 / 1           goto fail;
    1 / 1       if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                    goto fail;
    1 / 1           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:
```

We programmers should be humble people. Do you think we are? Anyway, why is that? Because we all fail at times, and shoot ourselves in our feet. I've done this myself many times. I've failed with raw pointers, with shared pointers, from lambdas, using the Linker and so on. I've even shot myself in the foot using Java bytecode! So I try to be humble about my abilities, I know I will fail again and again.

However, this is why I believe we need good tooling to assist us. Code coverage is one such tool, together with for example the clang sanitizers, good debuggers and so on.

It's also the reason why good methodology is important, through for example test driven development and unit testing.

More motivation

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

This is the outline of the rest of the talk. I will start with a short discussion of problems and limitations with traditional tools, then introduce kcov and demonstrate some of the main features.
I will then go into a bit of details about how kcov has been implemented, and give some lessons learned in the process.
Finally, I will discuss some other features of kcov, which are perhaps a bit less relevant on a C++ workshop.

Outline

Table of Contents

Most of you have probably encountered code coverage collection already, in some form. There are many variants of code coverage of increasing refinement, as you can see on the slide, and there are also others which are even more detailed and specialized.
In most settings, MC/DC coverage is the holy grail of code coverage, but also something which is incredibly hard to achive in tests. So does kcov instrumentation support MC/DC? Unfortunately, it's more of an unholy grail solution, focusing on statement coverage like most other collectors do.
So is code coverage collection effective? I've read a few studies, and the effect on software quality is mixed and unclear. That is, higher coverage percentage doesn't necessarily translate into fewer bugs. Interestingly, the type of coverage, from statement coverage and upwards, also doesn't make very much of a difference.

I believe code coverage has more to offer as a test development tool. Especially when developing unit tests it's useful simply to see if your tests miss something important, and it's the area where I personally use it most. Pretty much daily in fact.

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)

On UNIX, the traditional way of collecting code coverage has been to use gcov for collection, and lcov for HTML presentation, if desired. The process is a bit cumbersome, as can be seen in the example below. You first need a special –coverage option, which produces extra metadata files in the build directory. After running the program, you then get data for the actual execution in the build directory.

After this, you need to run the presentation tool, like lcov, to get actual output. If the process crashes during execution, you will lose the coverage output altogether. Gcov also doesn't gather code coverage from shared libraries.

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
```

Fortunately, using kcov does away with all these disadvantages, allowing collection without special compiler options, reporting without droppings and all done in a single step. Quite a sales pitch, don't you agree? We should note here that there are other collection methods, for example via clang and obviously a set of commercial tools.

Instead...

Table of Contents

My own background really started with bash coverage collection through shcov.
Kcov started as a fork of bcov, which doesn't rely on compile-time instrumentation of binaries, but instead uses the debug information present as long as you compile with -g. It can also produce lcov-like output.
I thought bcov was a great idea, but it was still somewhat cumbersome to use, separating collection and reporting. The codebase was a bit difficult for me to follow, so I forked it instead of improving on the original project. Today I would problably not have done it that way.
Guess the name! The original idea I had was to use kernel kprobes, which allows setting breakpoints on kernel code, while still retaining the userspace functionality. K therefore stands for kernel. The kernel functionality via kprobes never worked very well though.
Confusingly, there is now a kcov for the kernel. The kernel kcov is used to more intelligently guide syscall fuzzers.

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

My own background really started with bash coverage collection through shcov.
Kcov started as a fork of bcov, which doesn't rely on compile-time instrumentation of binaries, but instead uses the debug information present as long as you compile with -g. It can also produce lcov-like output.
I thought bcov was a great idea, but it was still somewhat cumbersome to use, separating collection and reporting. The codebase was a bit difficult for me to follow, so I forked it instead of improving on the original project. Today I would problably not have done it that way.
Guess the name! The original idea I had was to use kernel kprobes, which allows setting breakpoints on kernel code, while still retaining the userspace functionality. K therefore stands for kernel. The kernel functionality via kprobes never worked very well though.
Confusingly, there is now a kcov for the kernel. The kernel kcov is used to more intelligently guide syscall fuzzers.

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?

My own background really started with bash coverage collection through shcov.
Kcov started as a fork of bcov, which doesn't rely on compile-time instrumentation of binaries, but instead uses the debug information present as long as you compile with -g. It can also produce lcov-like output.
I thought bcov was a great idea, but it was still somewhat cumbersome to use, separating collection and reporting. The codebase was a bit difficult for me to follow, so I forked it instead of improving on the original project. Today I would problably not have done it that way.
Guess the name! The original idea I had was to use kernel kprobes, which allows setting breakpoints on kernel code, while still retaining the userspace functionality. K therefore stands for kernel. The kernel functionality via kprobes never worked very well though.
Confusingly, there is now a kcov for the kernel. The kernel kcov is used to more intelligently guide syscall fuzzers.

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)

This slide lists the main features of kcov. As I noted before, as long as you build with debug information, kcov has everything it needs for instrumentation. Optimized binaries can also be instrumented, but just like when debugging, this can sometimes give surprising results.
Kcov also automatically detects shared libraries and instrument them, both from compile time and loaded during runtime.
Unlike gcov+lcov, collection and reporting is done in the same command although kcov supports separating them if you collect coverage data from a target system where the source code isn't present. Output is both in HTML and various XML and Json formats to allow integration with external tools. More on that on the next slide.
When you run the same binary multiple times, kcov will accumulate the coverage data as long as the binary hasn't changed. Similarly, when collecting from multiple binaries with the same output directory, kcov will create a merged report with the sources from all binaries.

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

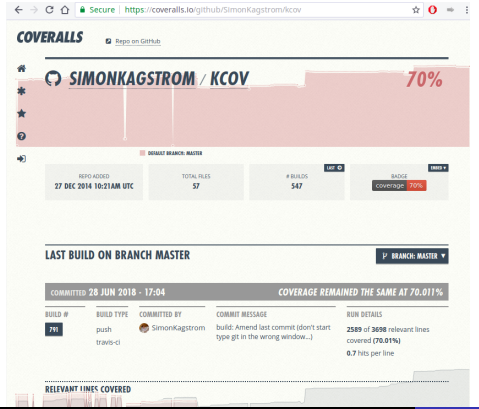
Kcov also integrates with several systems and sites for continuous integration. Jenkins has a plugin for Cobertura, normally a Java code coverage collection tool. As kcov produces XML output for Cobertura, it's easy to integrate in that environment. SonarQube is handled in a similar way.

It also supports some popular cloud services. NEXT. It can upload directly to coveralls, which is often used together with travis-ci. Coveralls is an easy way to get fancy web stats for your project coverage. NEXT. Similarly, codecov.io is also easy to integrate, but supports kcov directly from the upstream project.

Personally, I haven't picked a favourite cloud coverage service just yet, but instead use both with travis-ci and added dual badges on github.

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



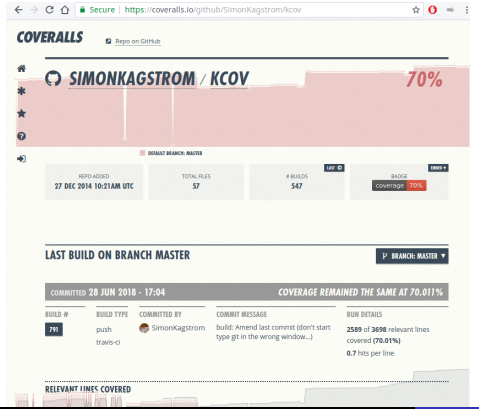
Kcov also integrates with several systems and sites for continuous integration. Jenkins has a plugin for Cobertura, normally a Java code coverage collection tool. As kcov produces XML output for Cobertura, it's easy to integrate in that environment. SonarQube is handled in a similar way.

It also supports some popular cloud services. NEXT. It can upload directly to coveralls, which is often used together with travis-ci. Coveralls is an easy way to get fancy web stats for your project coverage. NEXT. Similarly, codecov.io is also easy to integrate, but supports kcov directly from the upstream project.

Personally, I haven't picked a favourite cloud coverage service just yet, but instead use both with travis-ci and added dual badges on github.

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



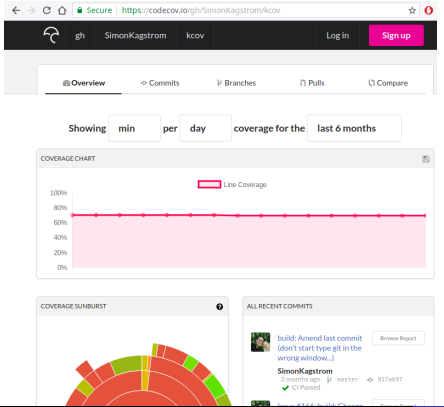
Kcov also integrates with several systems and sites for continuous integration. Jenkins has a plugin for Cobertura, normally a Java code coverage collection tool. As kcov produces XML output for Cobertura, it's easy to integrate in that environment. SonarQube is handled in a similar way.

It also supports some popular cloud services. NEXT. It can upload directly to coveralls, which is often used together with travis-ci. Coveralls is an easy way to get fancy web stats for your project coverage. NEXT. Similarly, codecov.io is also easy to integrate, but supports kcov directly from the upstream project.

Personally, I haven't picked a favourite cloud coverage service just yet, but instead use both with travis-ci and added dual badges on github.

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

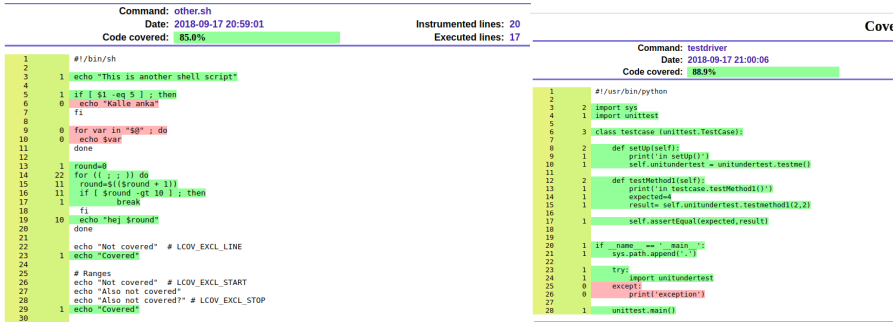


Kcov also supports Python and Bash coverage collection. Judging by the number of reported github issues, Bash coverage collection is surprisingly popular. Or the kcov implementation is just buggy!

Python and bash are a bit different that C/C++ collection, in that it reports line execution counts and not only whether a statement has been executed or not. Obviously, the implementation underneath is also quite different, but I'll disuss that a bit more later.

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()
```

- Demo: Make sure /tmp/kcov is empty
- kcov /tmp/kcov projects/build/kcov/target/src/kcov
- Show result in browser
- Show how a single file looks, describe 1/3, execution order etc
- Show file list again, note /usr/include etc
- Remove /tmp/kcov. All output is placed in the out-directory, so this cleans up everything from the coverage run
- Run kcov with –include-pattern, note that –exclude-pattern and paths also exist
- Run kcov unit tests, show merged report
- Discuss –collect-only, –report-only

Interactive Demo!

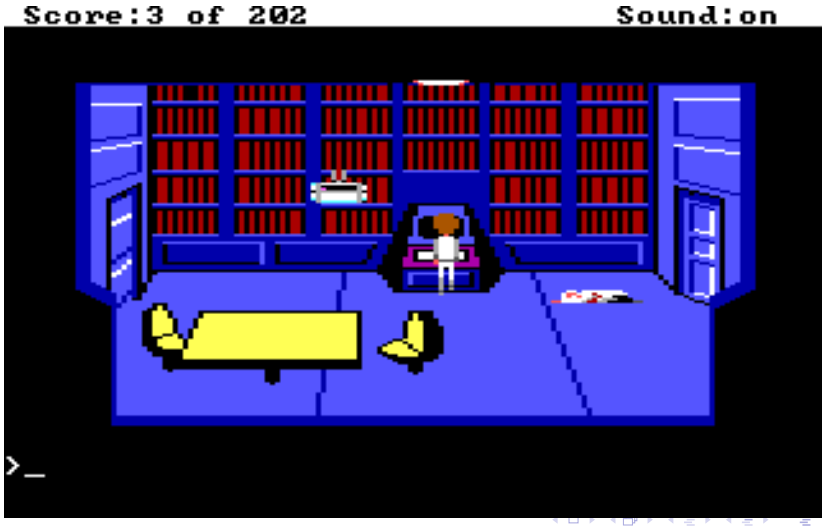


Table of Contents

Lets start with an overview of technologies used by kcov. Binaries in Linux are nowadays almost always stored in the ELF binary format. ELF binaries contain sections which contain code, data, constants, relocation information and so on. Where relevant, the sections contain a memory address where the code or data is loaded into memory during execution.
On Mac OS X, Mach-O is used as the binary format, and functions in pretty much the same way as ELF. Windows has some other format.
The most important of these for kcov is Dwarf, however. Dwarf is a format for debug information, and is related but independent from the binary formats. Dwarf sections are present in both ELF binaries, and Mach-O dittos and contain information needed by debuggers. For example, the mapping between source lines and addresses is needed when you set a breakpoint on a source line, variable type information when you print variables and so on.
I guess you've noted by now that the world of binary file formats and debug information is full of funny names. Stabs is an older format for debug information, seldomly used today.

We'll continue with how kcov uses these on the next slide.

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

Kcov relies on Dwarf debug information to set a breakpoint on each executable line in the program. The debug information contains records of file:line to address mappings, which is exactly what kcov needs.
NEXT. In the disassembly output, you can see this as the instructions marked in red, which correspond to DWARF entries.
Kcov starts the program in stopped state, sets all breakpoints and then lets the program continue again. On each executed breakpoint, kcov marks the file:line pair as executed, removes the breakpoint and lets the program continue again. That's the basic way kcov works.
Performance.

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>
```

Kcov relies on Dwarf debug information to set a breakpoint on each executable line in the program. The debug information contains records of file:line to address mappings, which is exactly what kcov needs.
NEXT. In the disassembly output, you can see this as the instructions marked in red, which correspond to DWARF entries.
Kcov starts the program in stopped state, sets all breakpoints and then lets the program continue again. On each executed breakpoint, kcov marks the file:line pair as executed, removes the breakpoint and lets the program continue again. That's the basic way kcov works.
Performance.

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>
```

Kcov relies on Dwarf debug information to set a breakpoint on each executable line in the program. The debug information contains records of file:line to address mappings, which is exactly what kcov needs.
NEXT. In the disassembly output, you can see this as the instructions marked in red, which correspond to DWARF entries.
Kcov starts the program in stopped state, sets all breakpoints and then lets the program continue again. On each executed breakpoint, kcov marks the file:line pair as executed, removes the breakpoint and lets the program continue again. That's the basic way kcov works.
Performance.

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>
```

So how does kcov know where to set breakpoints? Well, if you look into a process using readelf, you can see that it consists of multiple sections, some of which contain constants, some which contain data (variables), and some of which contain executable code.
Some of these are loaded into memory when the program is started, at an address you can see under the Address column on the slide. The debug information points into these sections, so with the debug info, you can just set breakpoints at the addresses found in the debug information, right?
Well, in some cases this is true, but not always. To start with, shared libraries are loaded on different and a bit random locations. They are so called PIEs, position independent executables. Entire programs can also be PIEs, to aid with a security feature called address space randomization.
Therefore, kcov needs to lookup where the binary sections are actually loaded in memory and adjust the debug info addresses according to that.

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
  [...]
```

There is no fully standardized way of setting and controlling breakpoints on UNIX. The ancient ptrace is typically used for this, through PEEKTEXT/POKETEXT options. Ptrace allows a controlling process to catch the signals from the controlled process when it hits breakpoints.
The original bcov implementation is the base for the Linux implementation, but Alan Somers have more recently extracted the OS-dependent parts and ported Kcov to FreeBSD. On Linux and FreeBSD, libelf and elfutils is used for parsing binaries.
OSX works in an entirently different way though. The key difference between OSX and Linux/FreeBSD is that OSX uses it's own binary format, Mach-O. So we now have Dwarves, Elves and Mach-O. For ELF, we have libelf to do the parsing, but Apple doesn't publish a library for Mach-O. I therefore opted on a simpler solution on OSX. The LLDB debugger comes with the development environment on OSX, and it conveniently offers a nice C++ API.
So on OSX, the entire parsing, process control and breakpoint setting is only 486 lines of code. The disadvantage with the LLDB implementation is that it's significantly slower than using regular ptrace. During normal debugging, you typically don't set tens of thousands of breakpoints!
Ptrace is quite messy to work with. The code snippet shown is for dumping the register file when the program stops at a breakpoint. As you can see, it's not standardized between architectures, not to mention between OSes. When you have the registers, kcov needs to know the program counter, which is one of the entries in the register file. Obviously in different positions on different architectures! The easiest way to find it has been to look in GDB or the Linux kernel source code.

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 affecting kcov?

So are there any interesting bugs?

There are some interesting quirks and bugs concerning Dwarf usage in Linux. The most irritating and interesting one is when the file:line to address entries contain invalid entries. We again have the disassembly output you saw before. NEXT. However, if you look at the second entry marked in red here, you see that it points into the middle of an instruction.
This is possible on x86 since it has variable instruction length, so an instruction can start on any byte address. Now, a breakpoint on x86 is simply a special instruction, encoded with a single byte, 0xcc hex. Setting a breakpoint therefore means overwriting the start of the instruction you want to break at with 0xcc. But if 0xcc is written into the middle of an instruction, you either get an invalid instruction, or even worse, a valid but unintended instruction.
kcov works around this by simply disassembing the entire program, and discarding Dwarf entries which doesn't point to the start of an instruction.
This is somewhat expensive, so it's only enabled with an option. On architectures which have a fixed instruction length, it instead discards entries which are unaligned to natural instruction borders.

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>
```

There are some interesting quirks and bugs concerning Dwarf usage in Linux. The most irritating and interesting one is when the file:line to address entries contain invalid entries. We again have the disassembly output you saw before. NEXT. However, if you look at the second entry marked in red here, you see that it points into the middle of an instruction.
This is possible on x86 since it has variable instruction length, so an instruction can start on any byte address. Now, a breakpoint on x86 is simply a special instruction, encoded with a single byte, 0xcc hex. Setting a breakpoint therefore means overwriting the start of the instruction you want to break at with 0xcc. But if 0xcc is written into the middle of an instruction, you either get an invalid instruction, or even worse, a valid but unintended instruction.
kcov works around this by simply disassembing the entire program, and discarding Dwarf entries which doesn't point to the start of an instruction.
This is somewhat expensive, so it's only enabled with an option. On architectures which have a fixed instruction length, it instead discards entries which are unaligned to natural instruction borders.

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>
```

The OSX port has a completely different set of problems. The most interesting one is perhaps handling of forks.
As many of you probably know, forking on UNIX means creating a new process, which is identical to it's parent through a weird system call which returns different values in the parent and child process.
Now, as we saw before, setting a breakpoint means modifying the instructions in memory. When a breakpoint is hit, the controlling process (the LLDB debugger in this case) is signalled and can take action.
However, LLDB doesn't actually attach to children as they are created via fork. So a breakpoint hit in the child will instead crash the process!
NEXT. So how can we workaround this in kcov?
Well, the workaround only goes half-way, since it doesn't cover the forked child just yet, but at least the program doesn't crash. What kcov does is to set a special breakpoint at fork, and when a fork happens, it will disable all breakpoints. After that, it will set a temporary breakpoint at the point where the parent returns and let the program continue.
When the parent returns again, it will hit the temporary breakpoint and all breakpoints can be enabled again. This is really a LLDB bug, which GDB doesn't have through it's follow-fork-mode.

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);
}
```

The OSX port has a completely different set of problems. The most interesting one is perhaps handling of forks.
As many of you probably know, forking on UNIX means creating a new process, which is identical to it's parent through a weird system call which returns different values in the parent and child process.
Now, as we saw before, setting a breakpoint means modifying the instructions in memory. When a breakpoint is hit, the controlling process (the LLDB debugger in this case) is signalled and can take action.
However, LLDB doesn't actually attach to children as they are created via fork. So a breakpoint hit in the child will instead crash the process!
NEXT. So how can we workaround this in kcov?
Well, the workaround only goes half-way, since it doesn't cover the forked child just yet, but at least the program doesn't crash. What kcov does is to set a special breakpoint at fork, and when a fork happens, it will disable all breakpoints. After that, it will set a temporary breakpoint at the point where the parent returns and let the program continue.
When the parent returns again, it will hit the temporary breakpoint and all breakpoints can be enabled again. This is really a LLDB bug, which GDB doesn't have through it's follow-fork-mode.

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

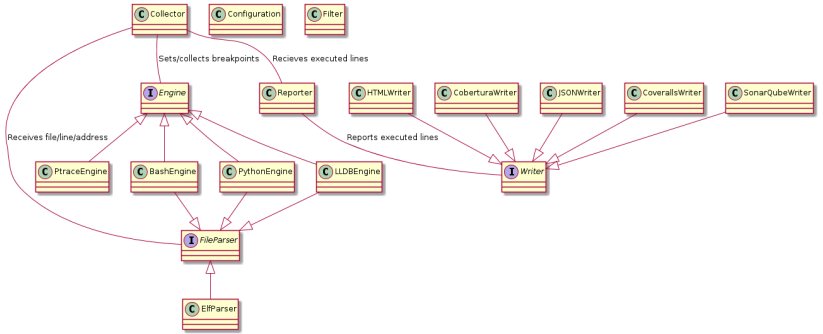
```
auto child = fork();
if (child < 0)
      printf("Error\n");
else if (child == 0)
      printf("In child\n"); // Breakpoint here is bad news!
else
{
      printf("In parent, child %d\n", child);
      wait(&status);
}
```

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

Kcov is written in C++, but unfortunately it's C++03, so a bit unmodern for these kinds of meetings! I'll just shortly discuss the kcov design a bit.
The design revolves around a set of abstract interfaces, as show on the figure and through heavy use of the observer pattern. Binary files are parsed via the IFileParser interface. Binary execution is handled through the IEngine interface, and output is handled via the IWriter interface.
Control is handled through the collector, which receives file/line to address mappings from the file parser. The collector will then set breakpoints, execute the program, and collect breakpoint hits through the engine. The reporter receives and memorizes executed lines and reports them to the writers, which produces the output. Pretty simple, don't you think?
The design was originally made to accomodate ELF and ptrace, and the only true File parser is the Elf parser. When kcov is used with Python, bash or indeed on OSX via LLDB, parsing and execution is kept together, so these implement both the engine and the file parser interfaces.

Design



Questions and comments!

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