Fuzzing driven code auditing and vice versa
Coverage analysis for better fuzzing
Hello! I'm Aleks, currently working as a security researcher at Cisco Talos VulnDev team.

Among other things, our team mostly works on analyzing vulnerabilities in 3rd party software and developing novel tools and techniques for finding said vulnerabilities. We tend to publish all of our stuff so you can check out some of our work on either of our github pages or on our main website.
What am I actually going to talk about today?

What's with the title? I meant it as a sort of workflow when auditing code with limited time. The main idea is to let fuzzing guide what you manually audit and let your auditing lead you to writing better fuzzing harnesses for your target. Also, while I am releasing the tool with this talk, the tool itself is fairly simple so I want to focus more on the insights gained from using it.

We'll go through a really quick overview of story so far. Including guided fuzzing, grooming corpuses and associated limitations. We want to see if intuitive limitations are actually true and how we can solve them. In order to do so, we must first define the questions. And to answer them, I've developed a simple analysis tool to help us. At the end, after presenting the tool, I will make some observations and insights gained from using the tool over several targets.
As I mentioned, let us quickly breeze through prior work and motivations. As I'm sure this audience is mostly familiar with the topic, I don't want to delve too much on it.
Long story short, the guy on the right wrote a tool on the left and there was much rejoicing.

Among many clever things that AFL does good, a couple stand out.

1. Compile time instrumentation → fast execution
2. Bitmap for fast tracking of code coverage of blocks with random IDs
3. Each block has a “counter” which effectively tracks state.

In essence, it actually tracks state transitions, rather than simple coverage. All that in a fool-proof, easy to use package. It broke, I think I can say, literally everything.
Apart from AFL, it's been a known fact for a while that the success of your fuzzing depends largely on the starting input corpus that you throw against the target. Even before AFL, corpus distillation was a right thing to do. It consists of reducing your corpus to contain only interesting testcases and throwing away the repetitive ones. You want to have corpus that covers as much of the target functionality as possible, but is optimized to small size and speed.

With AFL, now you suddenly have an option to improve your starting corpus, as you have feedback from your target telling you which mutated testcases to keep. So grooming the corpus became the main thing. Term Corpus driven fuzzing or fuzzing without fuzzing was coined by Ben Nagy. What you would do is take your corpus and fuzz one target with AFL letting it improve it by reaching new functionality and improving coverage. Then, you take that corpus and throw it against another target (possibly one you don't have source for) and see what sticks.
BTW, I've just recently published a small utility which might be helpful for this called crashdog. It just takes all the files from one directory, throws them against the target and keeps the crashing ones. Nothing fancy, but does include some useful sanity checking, check it out at this link:

https://github.com/Cisco-Talos/crashdog
What soon followed...

- AFL without source
  - AFL-Qemu (AndrewG)
  - Afl-dyninst (Me)
    - https://github.com/talos-vulndev/afl-dyninst
- fuzzing-project.org (Hanno Böck)
- Libfuzzer (Kostya Serebryan)
- DARPA Cyber Grand Challenge
- Google's OSS-Fuzz

In short, a whole lotta fuzzing.

Short on time:

In short, with AFL, Zalewski sparked a fuzzing renaissance. There's been a lot of development ever since and these are just some of the highlights.

There's been a whole lot of fuzzing.

Longer:

In short, with AFL, Zalewski sparked a fuzzing renaissance. People wrote tools that allowed you to fuzz binaries without source instrumentation. Andrew wrote AFL-Qemu and I've had a bit of fun using Dyninst for AFL-Dyninst which I've presented before.

Soon after, everybody was fuzzing all sort of stuff. Fuzzing project was started with some funding from Core Infrastructure Foundation as far as I can tell.

Also, a thing that really changed the way fuzzing is integrated into development lifecycle was libfuzzer. You were encouraged to write small, test-like, fuzzing harnesses for small parts of your code. There was also this whole deal of DARPA's Cyber Grand Challenge for which the authors from winning teams owe quite a lot of beers to Zalewski. Finally, with the size of Google's internal fuzzing infrastructure and accessibility of libfuzzer OSS-Fuzz happened. Now all of a sudden every open source project could get free fuzzing time on google's infrastructure. Many crashes were found and many bugs were fixed.

This in turn means that you need to be a bit more clever when you're fuzzing libraries and applications that have been fuzzed to death already by everybody, which is partially the motivation for this work.
Everybody seems to agree that:

*Evolutionary fuzzers get stuck.*

*Fuzzers don't trigger deep bugs.*

Let's try to confirm that.

That brings us to limitations of fuzzers.

Everybody seems to agree about these facts regarding fuzzers. 1. Generic fuzzers get stuck by complex conditions and 2. generic fuzzers don't trigger deep bugs that require complex states.

The goal of this research is to try and see those problematic parts of the code. To find places where fuzzer really got stuck. Parts of the code which weren't touched by fuzzing because state required to reach it is too complex to be hit by pure chance even with guided, evolutionary algorithm.
Meaningless comparisons to AFL

- Against deterministic mode for N hours
- "Unique" crashes

Fuzzing metrics are hard.

if short on time: A small sidenote about evaluating tools against AFL in papers. These are wrong things to do. Deterministic mode is slow by design and AFL's "unique" crashes aren't really unique. These two make naive evaluations against AFL basically meaningless. Fuzzing metrics are hard.

Before we continue, I wanted to make a small sidenote about different new fuzzers and potential AFL improvements. If you are publishing a fuzzing paper today, it's unfair to not evaluate your tool or proposed technique against AFL at all. But, at the same time, many publications I've gone through were guilty of these two errors.

First is running a comparison against vanilla AFL for, say, 24 hours and showing that the new tool performs better. These usually ignore the fact that AFL's first cycle can take much longer than others, that it's slow by design. It's doing deterministic fuzzing steps over all interesting inputs. In almost all cases, first, deterministic, cycle will have worse results than a comparable non-deterministic cycle. This is a design tradeoff made for completeness.

Second mistake is counting total number of crashes, or "unique" crashes as some sort of a meaningful metric. The problem is that AFL's definition of "unique" crash is quite different than what anyone actually looking for vulnerabilities would consider unique. AFL deems bug unique based on paths taken, where's traditionally one would consider unique bugs as ones having distinct root causes. This can lead to accidentally inflated numbers. Taking meaningful fuzzing metrics is hard...
Coverage analysis
What you should be doing

- Look at actual line coverage
- Gcov/Lcov is your friend

Even without any deeper analysis, just taking a quick look at actual line coverage that your corpus achieves against the target can quickly tell you if you are doing something wrong. It's easy to do.
There's a difference between AFL's sense of coverage and line coverage. AFL's coverage bitmap contains not only the fact that certain edge transition was executed, but, it will implicitly include some sense of process state. gcov on the other hand doesn't include this, it will just tell you if line was executed or not. OTOH, it will tell you how many times it was reached.

Both of these are useful and it's useful to combine them. Also, even though two total line coverage results (from two corpuses) seem similar (have similar percentage of code covered), one might be significantly better than the other because state complexity.
What are we interested in?

• Use coverage analysis to answer two questions:

What parts of the code aren't reached at all?
  e.g. hard-to-hit constants

What parts of the code we aren't hitting efficiently?
  e.g. failed checksums

With general coverage out of the way, we come to the actual questions.

What we want to know about our target and fuzzer is two things.

We want to see all parts of the code that the fuzzer should be reaching but isn't AND parts of the code that fuzzer isn't hitting efficiently. These are the actual limitations of generic fuzzers that we already mentioned.

A simple example of the first case would be a check for a large constant value or a magic word. If it's too large, a fuzzer has a very low chance of hitting it in a correct spot by pure chance.

Second case is a bit more tricky. It could happen that our corpus does indeed cover a whole lot of code, but because the mutator is dumb, only small number of generated testcases are actually considered valid by the target and others are rejected early. A simple example of this would be any form of checksum field in a file. Dumb fuzzer unaware of this would keep generating invalid testcases and would just waste time.

A classic example of this is PNG image file format and AFL even comes with an example patch to libpng that disables checksumming.

Wouldn't it be nice if we could find these problematic parts of the code automatically?
To do so, I wrote this tool which I dubbed "covnavi" short for coverage navigation because the idea is for the tool to guide you to problematic parts of the code.

What we want it to do is combine coverage data that we get from running a target over a corpus, and control flow analysis to pinpoint parts of the code that satisfy previously mentioned conditions.
What are we actually asking?

#1 Show all conditional statements where one branch is never taken

```c
164: 
165: 6: 
166: 6: 
167: 
168: 206: 
169: 
170: 
171: 0: 
172: 
173: 
```

#2 Show conditional statements where one is taken most of the time

```c
25887: 
25887: 
25887: 
25887: 
25887: 
23656: 
2281: 
```

How do those situations actually look in the code? In order to consider it useful, covnavi would need to be able to point out these two potentially problematic locations. First one, because fuzzier never got to this specific case in a switch and the second one because only a handful of testcases made it past this "if node" check and majority of them ended the parser/renderer early. If our tool can point out situations like these, we can consider it successful.

If we take a closer look at these screenshots we can see
How does it actually work? I used tools that I was familiar with to write this. I used gcov for gathering line coverage information which requires code that builds with gcc, I also use joern for control flow graph queries.
Gevalcov doesn't really need any introduction, but what's joern? Joern has been public for quite a while now though I'm not sure it's still under development.

Essentially, Fabs had this nice idea of taking all the source code, making a graph out of many properties and presenting it as a single graph database. He would lift all the source code and make a syntax graph, a Control Flow Graph, a Data Flow Graph, type graph and so on.

What you do then, is pick a node in the graph and ask questions about it and walk the graph. Is this a control flow node? Where can I go from here? Is this variable a constant? Where was it last accessed/assigned? and so on. It has a nice query language which allows you to basically script source code analysis. The way most people have been using it is to define some sort of anti-pattern of a vulnerability, encode that as a graph query and ask joern for all places in the source code where that type of pattern occurs.
Joern example

- Ask joern to give us all conditional statements
  
  ```java
  queryNodeIndex("type:IfStatement").id
  queryNodeIndex("type:SwitchStatement").id
  ```

- Get branches for each
  
  ```java
  g.v(IDX).out() // get condition, actual CFG node
  .filter{it.isCFGNode == 'True' & it.type == 'Condition'}
  .outE('FLOWS_TO') // follow "FLOWS_TO" edges
  .inV // get the nodes
  ```

if short on time: Without going into too many details, this is what joern queries look like. Find conditions, walk the graph to branches, take properties...

Let's just see a couple of joern examples. As I said, you analyze source code by making graph queries, although there are some shortcuts in it's query language.

In covnavi we execute something like this. First ask for all conditional statements. Then iterate over those to find branches. It's not very intuitive at first, but it's quite nice once you get a hang of it. You are basically constructing a path through a graph.

So, we start with an IfStatement, find its condition which is actually a CFG node and follow its "FLOWS_TO" edges to get to the branches.

You can make more complex queries to get if the branch was true or false and do all sorts of interesting things. This is the core of covnavi.
What it looks like?

Conditional(4968):

- code: if (!checkEncryption(ownerPassword, userPassword))
  - node id: 334836
  - location: ./poppler/PDFDoc.cc +286
  - branches: 2
  - True branch:
    - code: errCode = errEncrypted
    - node id: 334840
    - location: ./poppler/PDFDoc.cc +287
    - Is covered: False (0)
  - False branch:
    - code: catalog = new Catalog | this
    - node id: 334822
    - location: ./poppler/PDFDoc.cc +292
    - Is covered: True (3276)

Ok, what does covnavi give us?

There are a couple of steps to running this tool. First you must collect coverage, which might be fast or slow depending on your target and number of testcases. Then, you must import source code into net4j graph database to be used by joern. Depending on source size, it's fairly quick. With those two, you can use covnavi to combine coverage and joern info. Again, depending on number of conditional statements, this might take a while...

After that is done, you can just go through results in real time. All in all, total wait time is in order of tens of minutes, so I didn't bother to optimize any steps.

This is what we are presented as a result. We can see the code of the if statement, its location as well as both branches. For branches, we can see if each was covered or not and how many times. This is usually enough to make a quick glance and decide if the particular piece of code is interesting or not.

So, let's switch over to a ready db. The end result of the analysis is a JSON file containing stats for all conditionals. I intentionally kept it as json as it's easy to query and grep...

cat poppler.json | python -m json.tool | less

that's sometimes useful if you want to quickly find something... the navigation part of the tool comes next:

python covnavi.py show poppler.json poppler

The tool has a couple of nice options: filter by file name, mark as ignore, set threshold and so on. I'm working on adding diffing between two resulting databases so you can directly see the progress...

Ok, let's move on to results...
Main point of this talk. Can this actually answer our questions? Remember, first question was can we find points in code where fuzzer gets stuck. And yes we can.

Here we first see covnavi output saying that this if condition has always evaluated to true, and false branch was never executed. We have the locations and numbers of executions. This is in poppler code obviously. We can see that fuzzer never got to hit the font subtypes other than Type1 or MMType1 ... On the left we can see the actual code and it's coverage results. None of these other font types were ever hit. After seeing this, we might add those keywords to our dictionary and fuzz again. And then, after a while, we get that fuzzer actually did cover that code successfully! So, that's an easy win.
The second question, the question of "what code we aren't hitting efficiently" or "what kind of complex checks block the fuzzer" is somewhat harder to answer. First of all because if you only look at coverage of your corpus, you won't see it because AFL might not keep bad files around. Your corpus might look good, but then upon closer inspection, it turns out that 95% of the mutated files fail some early check and get rejected by the target. Looking at coverage of the corpus won't show this problem.

So, in order to try and find these kinds of problems, you want to take a sample set of fuzzed inputs. The bigger the better. Now, is it ok to just generate 100k random testcases by radamsa? Will that show all the same properties and problems that AFL might run into? In some cases yes, in some cases no.

Radamsa is less thorough than AFL, it might be that only some of the AFL's mutators get stuck...

Another problem is how to choose an optimal threshold. Should we look at conditionals where one branch is taken 99% of the times, or all from 70% and upwards? It depends on the code, and number of testcases executed. In general, with more testcases executed, these problematic spots will only be more pronounced and therefore easier to spot. Here we see two examples.

On the left, from libpng, there's a simple test for signed ints and we can see that only a handful of times this if was evaluated to false, so in most cases an error was thrown and parsing stopped.

Similarly, on the right, in mupdf font parsing code, a check for valid pdf stream is made and only a handful of calls actually passed the check.

These are fairly easy to spot with covnavi, although it's up to human to decide how to deal with them. For first one, you might write a simple fixer that makes sure no numbers are greater than PNG_UINT_31_MAX, but then again, that might not be so useful. That's the active auditing part...
Ok, we've seen that this can actually be useful in pointing out problematic code. Now I want to talk about some practical use and observations.
By fuzzing and using covnavi on a couple of different projects, we can identify different types of “blockers” that actually get fuzzers stuck. These are the situations to keep in mind and watch out for.

First one, no valid samples with given functionality is somewhat obvious, but covnavi will help you find it. If some part of the target has very specific requirements, evolutionary fuzzer will struggle to get to it. For example, even though AFL can pull jpeg out of thin air if given enough time, without a small sample it won’t be able to generate a PDF with an embedded jpeg any time soon. Identifying these situations is fairly easy because you can spot which big chunks of the code aren’t exercised at all.

As intuition tells us, hard-to-hit constants are also a blocker. There are other tools that try break up big comparisons into a number of small ones to try and solve this problem. But I’ve found that that can sometimes lead to coverage bitmap saturation and those should be used with care.

Then we have hard blocks, and what I mean by that is things like checksums or any sort of complex strong input validation.

One thing that might be obvious to everybody is that compiled target can have many differences depending on the compile time configuration. Debug builds vs release builds... Include 3rd party library for something or built-in alternative... For example, for a while Ubuntu was using jpeg decoder code that was built into poppler, instead of building it with libjpeg or whatever sane option... That decoder was unsupported and had a bunch of bugs. If you fuzz it in default configuration, you wouldn’t catch that...

Covnavi won’t point out these, but they are easy to spot by simply looking up coverage.

Also, if you have a custom fuzzing harness for some networking code which depends on config files... Invalid or incomplete configuration won’t get you far.
So, with problematic situations identified, what are some effective ways of getting past them? These are some well-known best practices.

**Simplest way to help** is better corpus of course. When you identify something that’s missing, make a custom small sample that has that functionality, add it to corpus and fuzz again.

Almost equally important as a good corpus is a good dictionary which in most cases easily solves the problem of large constants. This is also where iterative analysis is helpful. By rerunning coverage analysis after adding certain constants to the dictionary, you can easily check if that actually worked.

The hardest problem is hard blockers, and two ways of dealing with those are either modifying the target to remove them or adding custom fixers to fuzzers which actually fix the mutated testcases. Again, Ben Nagy had a tool that easily allows you to do that for AFL in languages other than C, called AFLFIX.

And lastly, for hard-to-reach code that depends on configuration or isn’t easy to reach by fuzzer otherwise, just audit the code manually. What else are you going to do while your fuzzers are running...
Now that we've seen that, let's look at my general workflow when using this approach.

First, we don't want to simply take some target, compile it and fuzz it with AFL in its default state, that will probably have too much overhead and will execute too much code we aren't generally interested in. Customize it, write a small harness that exercises just the interesting bits of code, just the actual attack surface.

This is enough to get started, and in some cases more than enough to get some decent results, but importantly this will also start making you a corpus...

After fuzzers seem to reach the plateau, it's a good idea to fire up coverage analysis and see what's going on. From those results you might want to decide to augment the corpus, augment the dictionary, change the harness or target... Anything to alleviate previously mentioned bad situations.

Then you fuzz again and keep doing those two steps until you are done.

One important thing, now that you've done some coverage analysis, is that you have some familiarity with the code. You know what code is actually reachable by the input, the code that actually makes up the attack surface, and that can lead you to interesting un-fuzzed functions that you might want to audit manually. That's where the "fuzzing driven code auditing" title comes from.

After doing that for a while, you hopefully have a pretty good corpus, so you can switch and focus on another target and so on.
An exploitable use-after-free vulnerability exists in the x509 certificate validation functionality in Apple macOS Sierra (10.12.3 release and 10.12.4 public beta versions) and iOS 10.2.1. A specially crafted x509 certificate can trigger a use-after-free vulnerability potentially resulting in remote code execution. In order to trigger this vulnerability the victim needs to visit a HTTPS website or other server which serves a malicious certificate or click on a file.

Fuzzing Without Fuzzing!

As an example of this... Sometime last year I wanted to hit x509 parsers. x509 being in ASN1 format ... Yeah, we've seen a fair number of problems there throughout the years. So, what I did was hit, one by one, every open source library that has it's own custom x509 parsers. That all fuzzing found a bug here and there BUT, after a few iterations of fuzzing, coverage analysis, fuzzing augmentation and fuzzing some more, I had this nice corpus of broken x509 certificates.

So, I took all these and used crashdog to throw them against macOS x509 parsing code to see what will stick and to my surprise that lead to this vuln. It's not the most easily exploitable UAF ever, but it's an interesting vector because it can easily be triggered remotely by just forcing a broken certificate in an TLS connection. Free bugs!
Sidenote: fuzzing network code

Custom harnesses > generic network AFL

Example: TALOS-2016-0219

An integer overflow in the process_binappend_prepend function which is responsible for processing multiple commands of Memcached binary protocol can be abused to cause heap overflow and lead to remote code execution.

I've seen a couple of project which try to bring fuzzing network services to AFL with various degrees of success. It's my opinion that you get better payback from again sticking with custom harnesses that basically hooks the protocol state automata then going and doing some form of network emulation to get AFL to work with it as is...

Granted, depending on the code and how tightly coupled it is to network primitives, in some cases this won't be possible, but has lead me to interesting bugs in the past.

Case in point, these memcached bugs. So again, spending some time to dig into the code, to write a better harness pays off...
Getting back to "experimental observations". Back to corpus driven fuzzing, the idea of reusing the corpus against a target which you can't directly properly instrument.

In essence, it works great for some targets. The success largely depends on the target file or protocol format. Irregular file formats (like PDF, or XML) can be implemented in many number of ways and fuzzing two different targets will yield uncomparable corpuses. Different parsers will focus on validating different things which can lead to a one target's otherwise good corpus being completely rejected by another.

As an example, poppler requires testcases with somewhat "sane looking" xrefs and would reject inputs early otherwise. On the other hand, mupdf will actively try to reconstruct the xref table... So simply taking a corpus generated by mupdf and throwing it at poppler will at first result in very little coverage...

That all being said, in most cases a preferred route of dealing with this would be to write fixers and preprocessors for mutated testcases, instead of directly modifying the target which in most cases won't be straightforward anyway.

Also, one thing to point out with AFL is it's use of "favored" paths. That means that AFL will test the input corpus to determine which files are interesting and which ones are not. Non-favorited ones will be skipped for the most part. It might be beneficial to, when switching to a new target with new corpus, force AFL to fuzz with all inputs, not just favored ones, in hopes that it might "fix" some.
And that brings us to the end. Code is available at this link, and it includes a somewhat detailed setup guide, mostly for a Java part. Joern is written in Java for some reason, and is painful to set up. Luckily, some kind soul has made it available as a Docker instance.

Thank you! And if you have any questions or suggestions, shoot me an email.