Kernel Fuzzing with syzkaller

BY MARK JOHNSTON

If you have ever been unlucky enough to fall victim to a FreeBSD kernel panic, you would be well-justified in asking just how those sloppy kernel programmers test their code. The kernel is the backbone of the entire system and changes to it should of course have been meticulously tested before users can boot up the latest and greatest build. In our defense, however, kernel programmers work in a harsh, inhospitable environment. The FreeBSD kernel is written in C, a programming language infamous for its subtle pitfalls and lack of amenities. The kernel also has to deal with several adversaries: first, it executes and provides services to all sorts of software, some of which may have malicious goals; second, it interacts with the computer's hardware and all of its associated warts, convoluted designs and outright bugs. Many kernel developers have spent sleepless nights debugging memory corruption that ultimately was the result of buggy device firmware that overwrites system memory when prodded a certain way. Finally, like any modern OS kernel, FreeBSD's makes use of all of the CPUs available in the computer, and kernel developers have to grapple with all of the intrinsic complexity of writing efficient, scalable and correct software for multi-core systems. In short, it's a tricky problem.

FreeBSD's developers put a great deal of effort into shipping stable, well-tested releases. It is worth thinking for a while about how one might test, say, a change to an existing system call, or a new system call. System calls are in a sense the front-end of the kernel: they provide the low-level abstractions used by all programs, and the invocation of a single system call may cause the kernel to execute thousands of lines of code on the invoker's behalf. A developer adding a new system call will certainly write some test programs to verify that it behaves according to its specification, but generally it is not possible to exhaustively test all possible inputs to a lone system call. Furthermore, test programs cannot prove the absence of a bug; even if the system call produced a correct result, a bug may have corrupted a piece of kernel memory in a way that is not detectable for a long time after the fact. System calls may also interact with each other: a multi-threaded program will often execute multiple system calls simultaneously, each updating some kernel state, so our hypothetical kernel developer must think carefully about the synchronization of these calls and how the hundreds of existing system calls might interact with the one in question.

These kinds of problems are not specific to kernel programming and we have many conceptual and technological tools that let us attack the stark complexity of writing bug-free kernel code, and deliver stable FreeBSD releases with confidence. Over the past several years a new such tool, syzkaller, has been extraordinarily successful at finding severe bugs in all major operating systems, including FreeBSD.

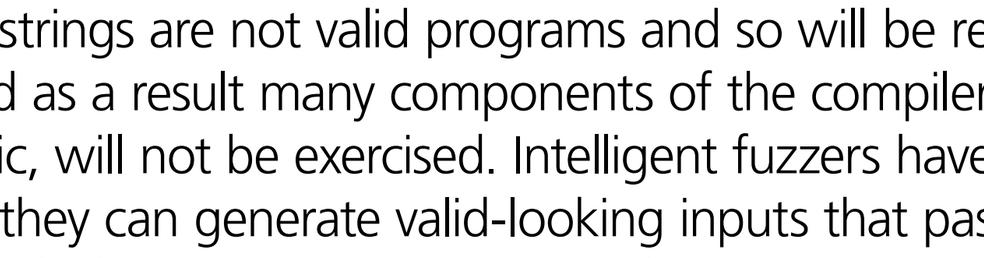
Coverage-guided Fuzzing

One important testing method for software that accepts untrusted input is fuzzing. Roughly speaking, fuzzing is the technique of programmatically generating inputs for the software under test, feeding that input to the software, and monitoring for unexpected results or side effects. This is an effective technique for finding bugs in the code that handles input validation, and has become an indispensable software testing tool. For instance your PDF reader, which you presumably use to open files found on the world wide web, will hopefully have been tested using a fuzzer among other things: the PDF specification is rather complicated and the code which parses it will be correspondingly so, making PDFs an attractive vector for malware authors. Indeed, fuzzers are often used by security researchers and malware authors to find security holes.

Fuzzing is one technique of many used to test software. One of its significant limitations is that it cannot generally verify that software is behaving correctly, only that it is not misbehaving according to some set of criteria. For instance, a fuzzer for a language parser would try to find input that causes the parser to crash, but the absence of a crash for a given input does not imply that the input was handled correctly according to the parser's specification. Fuzzers instead excel at finding corner cases and rarely executed code paths overlooked by other software testing methods and which are therefore quite likely to contain bugs. To maximize effectiveness, the software under test should use assertions and other forms of runtime checking to detect invalid states as early as possible.

Fuzzers vary in their level of sophistication. A naive fuzzer might generate purely random data and feed it directly to the software under test. While this approach may yield some fruit, it is unlikely to find anything other than very basic input validation bugs while consuming a large amount of computing resources. Consider a compiler fuzzer which simply generates random ASCII strings: most such strings are not valid programs and so will be rejected very quickly by the compiler's parser, and as a result many components of the compiler, such as optimization and code generation logic, will not be exercised. Intelligent fuzzers have some knowledge of the input format so that they can generate valid-looking inputs that pass basic verification logic. For instance, a fuzzer which aims to test an IPv6 packet processor would ensure that inputs at least start with the 4-bit version number that begins all valid IPv6 packet headers. It could achieve this by using a corpus of valid IPv6 packets as a starting point, or with some built-in knowledge of the IPv6 packet header layout, or likely some combination of the two.

A second effective optimization involves providing feedback to the fuzzer. A naive fuzzer would, in a loop, generate an input, feed it to the software under test, and wait for either a crash or graceful termination of the program. It has no general way to determine whether a given input helped improve test coverage of the software or not, and so cannot focus on "interesting" inputs. Consider a fuzzer target which performs input validation in two stages:



Stage 1 might simply verify that various components of the input have the correct length, while stage 2 verifies that the individual components contain valid values. If most input fails stage 1 validation, then stage 2 validation is left largely untested. However, if the fuzzer can

dynamically learn which inputs pass stage 1 validation, it can improve its coverage of stage 2 validation by prioritizing inputs known to pass stage 1.

There are multiple ways for a fuzzer to obtain feedback. For instance, it might measure the amount of time taken to process a given input and use a heuristic which discards inputs that are processed very quickly, under the assumption that such inputs are failing basic validity tests. Another technique, used by state-of-the-art fuzzing frameworks such as libFuzzer, AFL and syzkaller, measures code coverage. By leveraging software instrumentation facilities, a coverage-guided fuzzer can "trace" the code paths executed when processing a given input, and use that information to try and generate inputs which uncover previously unexecuted code. Fuzzers use this technique to achieve high levels of test coverage very efficiently, and indeed, the aforementioned fuzzers have been used to find thousands of severe bugs in all sorts of software projects, even those considered mature and well-tested.

syzkaller

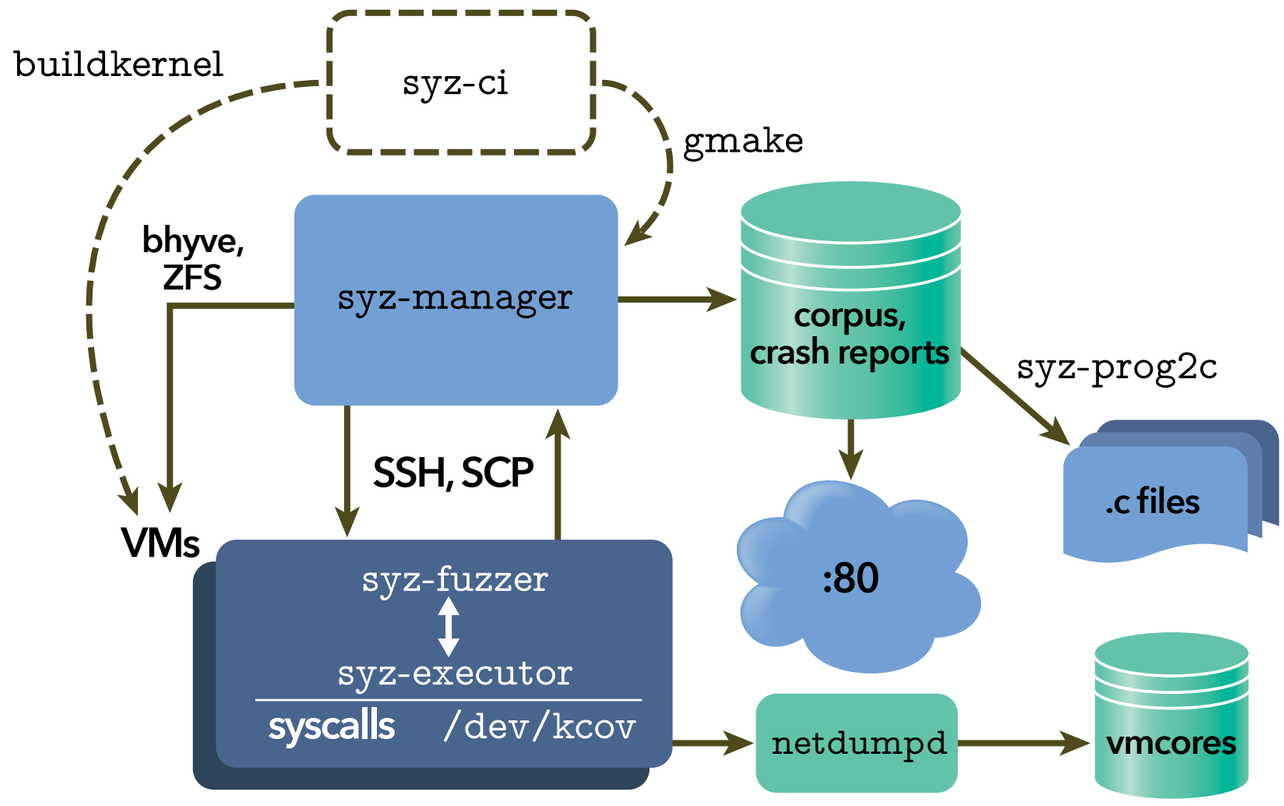
Operating system kernels handle input from a variety of untrusted sources: unprivileged processes will invoke system calls and may be trying to take control over the computer; a system connected to the internet processes network packets from untrusted sources; the kernel may be asked to mount a file system with invalid contents; a computer may support pluggable peripheral devices which can communicate directly with the kernel. In short, a useful kernel presents a massive attack surface, and years of high-profile kernel security holes show that there is much room for improvement among popular operatings systems. yzkaller seeks to improve this state of affairs.

syzkaller is an open-source coverage-guided kernel fuzzer by Dmitry Vyukov. It originally targeted Linux but has since expanded to support nearly a dozen other operating systems. syzkaller is sometimes described as a system call fuzzer but is flexible enough to target other operating system interfaces; for example, it has been used to fuzz Linux's USB stack and has found dozens of bugs in the USB subsystem alone. The details are complicated but the idea is simple: generate a program which invokes one or more system calls (or injects a packet into the network, etc.), run it, and check to see if the system diagnosed an error (for example by panicking). If not, collect kernel code coverage information and decide whether to try iterating upon the previous test program, or start anew. If so, collect information about the crash and try to discover a minimal test case that triggers the crash.

syzkaller is written mostly in Go and consists of a dozen or so loosely-coupled programs, all prefixed with `syz-`, that together provide a self-contained system to do all of the following:

- Start and run a set of operating system instances, typically in virtual machines.
- Monitor those virtual machines for crashes or other diagnostic reports, typically by monitoring console output.
- Generate programs to run under the target operating system, using coverage information to drive decisions about what to try next, and run them.
- Maintain a database of observed crashes and diagnostic reports, to try and classify distinct bugs found.
- Provide a web dashboard displaying statistics, code coverage information, and observed crashes and their reproducers if any.
- Periodically update itself and the operating system under test without any manual intervention.
- Attempt to bisect new crashes down to the commit introducing the bug.

The high-level components of this system as it might run on FreeBSD are depicted here:



Thanks to Google, the syzkaller developers provide numerous public syzkaller instances running in continuous integration mode, wherein syzkaller updates itself and the target operating system regularly. These "syzbot" instances find bugs in the latest builds of their targets, so regressions are reported quickly and completely automatically. The FreeBSD instances have found numerous bugs and reproducers, enabling developers to both diagnose and fix bugs quickly and to provide higher-quality releases.

kcov(4)

syzkaller is not the first kernel fuzzer but is undoubtedly the most prominent. Newsgroup posts from the early 1990s describe programs which bombard UNIX kernels with random system calls to great effect. Peter Holm's stress2 test suite for FreeBSD performs some targeted fuzzing of certain system calls (among many other things). However, syzkaller introduces a key innovation in its use of code coverage to drive test case generation. This makes use of the `kcov(4)` kernel subsystem, written also by Dmitry Vyukov for Linux but later ported to other operating systems by their respective developers. While syzkaller does not strictly require code coverage information, it is much more effective with this extra feedback from the kernel.

In FreeBSD, `kcov(4)` is a wrapper for LLVM's SanitizerCoverage. Sanitizers are compiler features which inject bits of code enabling certain types of introspection into the compiled result. For example, LLVM's AddressSanitizer inserts special function calls before every single memory access by the generated machine code; the calls can be used to determine whether the memory access is somehow invalid, for example because it corresponds to a use-after-free. This provides powerful bug-detection facilities similar to Valgrind but using different mechanisms: Valgrind works by running the unmodified target program in a software virtual machine which can intercept memory accesses and perform validation, whereas sanitizers are implemented by the compiler itself and require special compilation flags. Sanitizers and Valgrind both introduce significant performance overhead and are generally used only in testing scenarios. Sanitizers have the added benefit that they can sometimes be used to validate a kernel, while Valgrind cannot.

SanitizerCoverage inserts function calls according to the control flow of the generated code. Most CPU instructions do not modify control flow: once the instruction is completed, the CPU fetches and executes the subsequent instruction from RAM. Control flow instructions cause

the CPU to jump to a different address and begin execution there instead. This is how basic programming language constructs like if-statements, loops and `goto` work under the hood. A compiled program can thus be broken down into a set of "basic blocks," where a basic block is a sequence of non-control flow instructions. Following the end of each basic block is a control flow instruction. Then, if the goal is to figure out which pieces of code get executed in response to a given input, it suffices to trace out which basic blocks get executed.

SanitizerCoverage, roughly speaking, inserts function calls in between each basic block, as in this machine code for the FreeBSD kernel function `vm_page_remove()`:

```
<+0>:     push   %rbp
<+1>:     mov    %rsp,%rbp
<+4>:     push   %r15
<+6>:     push   %r14
<+8>:     push   %rbx
<+9>:     push   %rax
<+10>:    mov    %rdi,%rbx
<+13>:    callq  0xffffffff81167dc0 <__sanitizer_cov_trace_pc>
<+18>:    mov    %rbx,%rdi
<+21>:    callq  0xffffffff815b7c50 <vm_page_remove_xbusy>
<+26>:    mov    %eax,%r14d
<+29>:    mov    $0x1,%ecx
<+34>:    xor    %esi,%esi
<+36>:    mov    $0x2,%eax
<+41>:    lock cmpxchg %ecx,0x54(%rbx)
<+46>:    setne  %r15b
<+50>:    sete   %sil
<+54>:    xor    %edi,%edi
<+56>:    callq  0xffffffff81167ea0 <__sanitizer_cov_trace_const_cmp1>
<+61>:    test   %r15b,%r15b
<+64>:    jne    0xffffffff815b78f9 <vm_page_remove+73>
<+66>:    callq  0xffffffff81167dc0 <__sanitizer_cov_trace_pc>
<+71>:    jmp    0xffffffff815b790d <vm_page_remove+93>
<+73>:    callq  0xffffffff81167dc0 <__sanitizer_cov_trace_pc>
<+78>:    movl   $0x1,0x54(%rbx)
<+85>:    mov    %rbx,%rdi
<+88>:    callq  0xffffffff81107950 <wakeup>
<+93>:    mov    %r14d,%eax
<+96>:    add    $0x8,%rsp
<+100>:   pop    %rbx
<+101>:   pop    %r14
<+103>:   pop    %r15
<+105>:   pop    %rbp
<+106>:   retq
```

Here the `__sanitizer_cov_trace_*` function calls are inserted by SanitizerCoverage and can be implemented by the kernel. `kcov(4)` works by implementing these functions.

In typical usage, a user program allocates a buffer to store coverageinformation, opens `/dev/kcov` and uses `ioctl(2)` to map the buffer into the kernel and to enable tracing of the current thread. When the thread subsequently enters the kernel, perhaps to execute a system call, the coverage tracing hooks log the address of each basic block into the buffer. When the thread disables tracing, again using an `ioctl(2)` call, it can make use of the information provided in the buffer. For instance, the recorded addresses could be piped into the `addr2line(1)` program to find the file and line number of the traced C code. The `kcov(4)` manual page contains the details of this `ioctl(2)` interface as well as some example code.

syzlang

Earlier we pointed out that fuzzers work better when they have some knowledge of the software's input format, rather than treating it as a black box. While syzkaller could theoretically invoke system calls without any knowledge of what they do or what parameters they take — using only coverage information to try and "learn" which parameter values result in more code execution — this is both inefficient and potentially counter-productive. Consider what happens if a fuzzer invokes `kill(-1, SIGKILL)`: the kernel will do what it was asked to do and immediately kill the fuzzer process.

Unfortunately, system call interfaces cannot be discovered programmatically. In other words, there is generally no way to ask the kernel to describe the set of system calls that it implements. Even a set of C function prototypes omits many important details. Consider `read(2)`:

```
read(int fd, void *buf, size_t nbytes);
```

First, `fd` is here represented by an integer, but really must be a valid file descriptor as well. There are 4,294,967,295 possible values for `fd` and all but a tiny fraction of them are invalid. Second, it is not clear what the kernel is expected to do with `buf`: is the kernel supposed to read data from that address, or write to it, or both, or neither? Third, `nbytes` is supposed to represent the size of the buffer `buf` but the prototype gives no indication that these two parameters are related; the C language is simply not expressive enough to do so. If you are familiar with `ioctl(2)`, think for a bit about how it makes a bad situation even worse.

To solve these problems, syzkaller introduces syzlang: a language for modeling the kernel's programming interfaces. It is flexible enough to define data layouts that are binary-compatible with C types, and expressive enough to describe inter-related C parameters, among other things. In syzlang, the `read(2)` prototype above becomes:

```
read(fd fd, buf buffer[out], count len[buf])
```

Unlike C (but like Go), the parameter name comes first, followed by the type. Right away we can see that this definition provides more information than the C prototype: there is an `fd` type, to distinguish file descriptors from plain integers; `buf` is a pointer to a buffer mapped in the caller's address space, and the `[out]` annotation signifies that it is an "out-parameter," i.e., the kernel is supposed to write data to the buffer; `count` is the length, in bytes, of the buffer `buf`.

The use of specialized types to represent file descriptors and other kernel resources is important for generating programs that do "interesting" things since many system calls take as input the results of previous system calls. For example, to read data from a file a program might execute the following sequence of system calls:

```
const size_t len = 4096;
void *buf = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_ANON, -1, 0);
int fd = open("/tmp/foo", O_RDWR);
read(fd, buf, len);
close(fd);
munmap(buf, len);
```

Note that the results of the `mmap(2)` and `open(2)` calls are used as input to subsequent calls. Fuzzing `munmap(2)` and `close(2)` does not make much sense without earlier calls to `mmap(2)` and `open(2)`, so syzlang models the corresponding resources and syzkaller creates chains of system calls using these relationships to guide its choices.

The use of `mmap(2)` also illustrates a need to define the set of valid values for "flag" parameters such as the third and fourth arguments. syzlang has a built-in `flags` type for this:

```
mmap(addr vma, len len[addr], prot flags[mma_prot],
     flags flags[mmap_flags], fd fd, offset fileoff)

mmap_prot = PROT_EXEC, PROT_READ, PROT_WRITE
mmap_flags = MAP_SHARED, MAP_PRIVATE, MAP_ANON, ...
```

The fuzzer will chose zero or more flag values when creating an argument for a flag parameter.

syzlang can also use subtyping to more accurately model system call interfaces. Network connections and open files are both represented by file descriptors in the system call interface, but many system calls accept only certain types of file descriptors. For instance, one of `sendfile(2)`'s parameters is a file descriptor corresponding to a network connection on which to send a file's data. Such descriptors are created using `socket(2)` or `socketpair(2)`. Passing a descriptor for a regular file here will fail, so to save the fuzzer time we can define a new socket type. First we have the definition for `fd`:

```
resource fd[int32]: 0xffffffffffffffff, AT_FDCWD
```

This derives `fd` from the built-in integer type `int32` and defines a couple of special values: -1, for system calls where a parameter of type `fd` is optional (such as `mmap(2)`), and `AT_FDCWD`, used by `openat(2)` and similar system calls. Then we can define a derived resource type for sockets:

```
resource sock[fd]

socket(...) sock
sendfile(fd fd, s sock, ...)
```

A similar trick is used for system calls where layout of a parameter depends on the value of another parameter. `ioctl(2)` is the prime example of this, but `fcntl(2)`, `setsockopt(2)` and `bind(2)` behave similarly. For example, `pf(4)` defines a large set of `ioctl` commands, but each has its own argument type. We can describe them precisely in syzlang:

```
resource fd_pf[fd]
openat$pf(fd const[AT_FDCWD], file ptr[in, string["/dev/pf"]], ...) fd_pf
ioctl$DIOCRADDTABLES(fd fd_pf, cmd const[DIOCRADDTABLES], arg ptr[in, pfioc_table])
```

Here we define a new `fd` type which corresponds to a file descriptor for `/dev/pf`. Then, special "flavours" of `openat(2)` and `ioctl(2)` describe how the fuzzer can open a `pf(4)` device and issue the `DIOCRADDTABLES` `ioctl`, used by `pfctl(8)` to define an address table.

syzlang definitions are compiled at build-time into tables used by syzkaller's fuzzer. syzkaller maintains its own internal representation of system calls and their parameters, and its representation of a program is simply a list of calls and parameters. All fuzzing is performed using these representations; when a reproducer for a kernel bug is found, it is finally translated into a standalone C program. This can be done manually using `syz-prog2c` but this is typically handled automatically.

New system call definitions are added frequently since in most cases syzkaller is still playing catch-up: the existing kernel interfaces are massive and defining them in syzlang requires time and effort. In particular, many components of FreeBSD are not yet described by syzlang and therefore do not get tested by syzbot. Adding to the FreeBSD syzlang definitions is a great way to start contributing to the syzkaller project and to help ensure that FreeBSD gets as much test coverage as possible.

syz-manager

So far we have looked at the mechanisms by which syzkaller addresses the generic technical problems faced by all fuzzers: obtaining feedback from the target software (via `kcov(4)`) and describing kernel interfaces (with syzlang). Now we can look more at some of the machinery required to fuzz an operating system kernel.

A fuzzer's goal is to "exercise" the code being tested, so it needs an environment in which to execute the code and provide input. Fuzzing a kernel poses some extra challenges: the fuzzer needs to run in the same system as the kernel being tested, so if it achieves its goal and triggers a kernel panic, all of the fuzzer's state will be lost. syzkaller's solution is to run the target kernel in a set of virtual machines which can be safely wiped without losing anything important. These VMs can run on the same host as syzkaller or in cloud environments such as Google Compute Engine.

`syz-manager` is the main front-end program of syzkaller. It takes a configuration file as input and starts a number of VMs according to the configuration. It automatically installs and starts the fuzzer programs in each VM instance and communicates with them using an RPC interface over SSH. `syz-manager` also monitors the VM consoles to detect crashes. When a crash occurs, the VM is automatically re-created. VMs are restarted periodically even in the absence of a crash; one reason for this is to enable "corpus rotation."

`syz-manager` also maintains the instance's crash database. When a crash is discovered, `syz-manager` adds an entry to the crash database. Crashes are identified by the panic message printed by the kernel, and when a new crash is found, `syz-manager` dedicates a subset of the VMs to spend time attempting to reproduce the crash. Programs that were executed leading up to the crash are replayed, and if a crash can be reproduced, syzkaller also attempts to find a minimal reproducible for the crash to add to the crash database. Finally, syzkaller attempts to translate the crashing program into a standalone C program, making it easy for developers to debug crashes without needing a syzkaller installation on hand.

Most of the work to set up syzkaller involves creating a VM image containing the target operating system. The VM's kernel should have `kcov(4)` enabled, and an SSH key for the root user must be installed. The VM image is used as template; when syz-manager starts, it creates a snapshot of the image before starting VMs, and each VM uses a local copy of the template. When a VM is restarted, it gets a fresh copy of the template image, so any damage done by the fuzzer is discarded.

`syz-manager` supports a number of different hypervisors and cloud APIs. On FreeBSD one can use bhyve as the back-end hypervisor. To create a VM image template `syz-manager` uses ZFS clones, since bhyve lacks support for creating disk image snapshots. Upon starting up `syz-manager` also starts a web server, providing a dashboard containing statistics and code coverage information, deduplicated crash reports, and crash reproducers. A sample `syz-manager` configuration looks like this:

```
{
        "target": "freebsd/amd64",
        "http": "0.0.0.0:8080",
        "workdir": "/data/syzkaller",
        "image": "/data/syzkaller/vm.raw",
        "syzkaller": "/home/markj/go/src/github.com/google/syzkaller",
        "procs": 4,
        "type": "bhyve",
        "ssh_user": "root",
        "sshkey": "/data/syzkaller/id_rsa",
        "kernel_obj": "/usr/obj/usr/home/markj/src/freebsd/amd64.amd64/sys/SYZKALLER",
        "kernel_src": "/",
        "vm": {
                "bridge": "bridge0",
                "count": 32,
                "cpu": 2,
                "hostip": "169.254.0.1",
                "dataset": "data/syzkaller"
        }
}
```

This configuration specifies 32 VM instances; in general, more VMs is better since each VM runs test programs in parallel with the others. The "cpu" parameter defines the number of virtual CPUs given to each VM, and the "procs" parameter defines the number of fuzzer processes that will run in each VM. Having multiple virtual CPUs and fuzzer processes improves the odds of finding certain types of bugs, but over-subscribing the host may decrease the effectiveness of fuzzing by making it hard to reproduce bugs. It is reasonable to configure one or two virtual CPUs per host CPU, but more than that is probably too many.

See the FreeBSD/syzkaller documentation for details on how to build and configure your own syzkaller setup. With a configuration file written, syzkaller can be started with:

```
# syz-manager -config /path/to/config
```

When using bhyve, syzkaller needs to run as root in order to create virtual machines. It is possible to run syzkaller in a jail with some effort; some ongoing work aims to make this simpler.

If you wish to run your own private syzkaller instance, do be prepared to be patient — now that much of the low-hanging fruit has been fixed, it can take days for syzkaller to find a new kernel bug.

Fuzzing the Kernel

Now that we have encountered most of the machinery that syzkaller uses to fuzz operating system kernels, we are equipped to start looking at the brains of syzkaller.

Aside from the crash database, syzkaller's main piece of persistent state consists of the corpus: a representative set of programs whose execution generates coverage of the kernel. The corpus — initially empty — is effectively a seed for the fuzzer: a new test program is generated by taking a program from the corpus, mutating it in some small way, and checking to see if previously uncovered kernel code was covered by the new program. If so, the program may be added to the corpus and subsequently used as the starting point for other programs. Algorithmically, the fuzzer does nothing except try to increase the size of the corpus. Many heuristics are applied to try and make this more effective for syzkaller's real purpose — finding bugs — but the core idea is very simple.

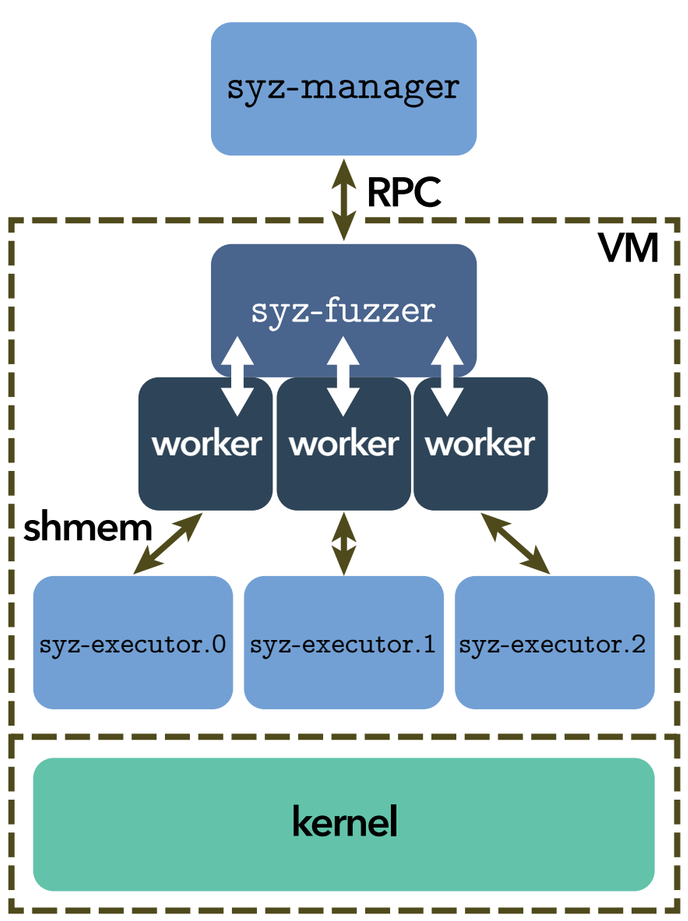
Several types of program mutations are possible. The fuzzer might:

- splice several programs together
- insert a new system call
- remove an existing system call
- modify one of the parameters to a call in the program

If the corpus is empty, the fuzzer will create a new program by generating a random list of system calls with randomly selected arguments. This is also done periodically even when the corpus is non-empty.

Inside each VM managed by syzkaller runs a pair of programs, `syz-fuzzer` and `syz-executor`, which communicate using a shared memory interface. As their names suggest, `syz-fuzzer` generates test programs and `syz-executor` actually executes them. `syz-fuzzer` and `syz-manager` coordinate using a simple RPC protocol; since `syz-fuzzer` generates programs which may crash the VM, it relies on `syz-manager` to store the corpus.

`syz-fuzzer` is started by `syz-manager` over SSH. When it begins, it establishes an RPC connection with `syz-manager` and creates a number of work queues, each of which is managed by a thread (really, a goroutine). The worker threads each spawn a `syz-executor` instance and immediately begin fuzzing, yielding the following picture:



Worker threads perform most of the work of adding to the corpus: they generate new programs and mutate existing ones. They also handle special types of work:

- Triage: when a program appears to generate new coverage it is placed in the triage queue for further refinement. The triage step tries to determine whether the program behaves consistently (i.e., re-runs generate the same coverage info), and if so, tries to minimize the size of the program while maintaining its coverage.
- Smashing: when a program has been triaged and appears worthy of being added to the corpus, the worker spends extra time mutating it to look for new coverage.
- Candidate processing: `syz-manager` may send candidate programs to the fuzzers in some cases. The worker executes them, potentially creating triage or smash work.

In steady-state operation, `syz-fuzzer` uses two RPCs to communicate with the manager: `Poll` and `NewInput`.

`Poll` is invoked periodically to update the fuzzer's snapshot of the corpus and global coverage information, and to collect candidate programs for fuzzing. It also serves to re-process the existing corpus when syzkaller starts up; a typical syzkaller installation will periodically update itself and the target kernel, and must subsequently restart. The saved corpus is immediately distributed among the fuzzers for execution and triage since the updated kernel may handle existing corpus items differently from when they were last evaluated.

`NewInput` is used to send triaged programs back to `syz-manager` as possible candidates for the global corpus. `syz-manager` will reject new inputs in some cases, for instance to avoid blowing up the size of the corpus, or if another fuzzer had already discovered a similar program. If accepted, new corpus programs eventually become visible to other fuzzer instances via `Poll`.

Unfortunately, code coverage is not an ideal metric: 100% code coverage of a program does not preclude the existence of detectable bugs, especially in multi-threaded code such as a modern operating system kernel. Optimizing for an imperfect metric tends to yield suboptimal results — we (hopefully!) do not evaluate programmers based on the number of lines of code they have written. In syzkaller's case, valuable test programs may be discarded if they do not add to the corpus' code coverage. To try and alleviate this problem, syzkaller performs corpus "rotation": some system calls and corpus programs are hidden from individual fuzzers to force them to find programs with equivalent coverage but hopefully new characteristics. This can result in duplicated effort but helps to ensure that the system does not become "stuck" by finding local maxima.

Program Execution

To round off our examination of syzkaller a look at `syz-executor` is in order. syzkaller uses an internal representation of system call programs for the purpose of fuzzing, but of course has to actually run them somehow. `syz-executor` is the component of syzkaller that performs this task; unlike the rest of syzkaller, it is written in C++.

The executor is spawned by `syz-fuzzer` worker threads and uses a simple shared memory interface to communicate with the worker. It first creates a pool of threads to actually execute system calls, and then opens `/dev/kcov` and uses `ioctl(2)` to enable collection of code coverage information that is returned to the worker. Quite a lot of additional initialization may happen at this point, depending on how syzkaller is configured. For instance, the executor may enter a software sandbox in an attempt to limit the effects of the test program: a program which sends signals to unsuspecting processes is likely to wedge the VM and trigger a

costly timeout and restart. It may also initialize devices or network facilities as part of a targeted fuzzing regime.

When it comes time to execute system calls, `syz-executor` iterates over the call list and assigns an idle thread to each one, waiting for threads to become free if necessary. Initially, the main thread waits for a short period after each call is dispatched. Once the input program has finished, it is executed a second time in "collision mode": rather than waiting for a short period after each call is dispatched, pairs of system calls are allowed to execute concurrently, helping to trigger race conditions in the kernel that would otherwise be left unexercised.

Actual system call execution is achieved using the handy `syscall(2)` system call, a generic system call which takes a system call number and variable list of parameters as arguments. Internally the kernel uses the system call number to route the call to the requested handler. The system call's result is also recorded for use in prioritizing and triaging programs: a successful system call is weighted more favorably than a failed system call.

Conclusion

If you managed to get this far, please don't stop here! syzkaller is the subject of quite few talks, articles and even research papers — check out syzkaller's documentation for some curated links. This article only scratches the surface of syzkaller's internals, and the sources are as usual the authoritative reference on how syzkaller actually works.

Fuzzing is a fascinating subject and there is a certain thrill to watching a fuzzer in action — particularly when it finds bugs in your favorite operating system. We encourage you to give it a try.

MARK JOHNSTON is a contractor and FreeBSD src committer based in Toronto, Canada. He is particularly interested in kernel debugging and in finding new ways to help improve the stability of FreeBSD. In his spare time he enjoys cooking, playing Bach's cello suites, and impeding his productivity by experimenting with custom keyboard layouts.