perf_fuzzer: Exposing Kernel Bugs by Detailed Fuzzing of a Specific System Call (2019 Update)

Vincent M. Weaver
University of Maine
vincent.weaver@maine.edu

Dave Jones
davej@codemonkey.org.uk

ABSTRACT
Fuzzing is a process where random, almost valid, input streams are automatically generated and fed into computer systems in order to test the robustness of user-exposed interfaces. While many fuzzers generally target the entire Linux system call interface, we instead target a specific system call and find hard-to-find bugs that would take much longer to find with a purely random search.

The `perf_event_open()` system call was introduced in 2009 and has grown to be a complex interface with over 40 parameters that interact in subtle ways. By using detailed knowledge of typical `perf_event` usage patterns we develop a custom tool, `perf_fuzzer`, that has found bugs that are more generic, system-wide, fuzzers have missed. Numerous crashing bugs have been found, including a local root exploit as well as an issue introduced by the KPTI meltdown workaround. Fixes for these bugs have been merged into the main Linux source tree.

Testing continues to find new bugs, although they are increasingly hard to isolate, requiring development of new isolation techniques and helper utilities. We describe the development of `perf_fuzzer`, examine the bugs found, and discuss ways that this work can be extended to find more bugs and cover other system calls.

1. INTRODUCTION
Fuzzing is an automated method of finding bugs and security issues. A fuzzer stresses various aspects of computer systems by methodically generating inputs designed to trigger boundary conditions that may not have been well tested. Nearly-correct inputs are generated and fed into the software to see if errors are handled correctly; if not the program may crash, or worse, lead to exploitable security issues.

Fuzzing can be done at any level of the computing stack, from high-level user programs down to the underlying hardware implementation.

Fuzzing has a long history. Many in academia consider it a “solved” issue with only “incremental” improvements and thus not worthy of investigation. However we show that an incremental targeted fuzzer can still find large numbers of critical bugs. The topic may be solved, but it is apparently not solved very well.

We fuzz the highly visible and often-fuzzed Linux kernel codebase and find that by using domain knowledge of a specific complex system call we can quickly uncover a large number of security issues.

The `perf_event_open()` system call is the primary entry point into the Linux kernel's `perf_event` performance monitoring subsystem. Introduced in 2009, the `perf_event` interface allows creating a file descriptor that is linked to various types of system performance measurements: software events maintained by the kernel (page faults, context-switches), hardware events maintained by low-level hardware counters in the CPU (cache misses, instruction counts), and other performance information that fits the interface (such as RAPL energy measurements on Intel CPUs).

The `perf_event_open()` call has grown to be a complex interface which takes a structure with over 40 members that interact in subtle ways. Various other system calls interact with `perf_event` file descriptors, providing a large surface for potential errors.

We write a syscall-specific fuzzer, `perf_fuzzer`, that automatically tests this interface. To date at least seventeen major bugs have been found and fixed; most are denial of service (DoS) bugs that can crash a system, but at least one is a local root exploit. Fixes for all of these bugs have been contributed back upstream to the main Linux kernel source tree. Testing continues to find new bugs, although they are becoming more obscure and harder to isolate and fix.

On modern systems `perf_event` is disabled by default, making it less of a security target than other interfaces. The reason it is now disabled by default is primarily due to the large numbers of bugs found by the `perf_fuzzer`.

2. RELATED WORK
The use of random inputs when testing computer systems has a long history, although at times it has been considered less effective than more formal testing methods. Duran and Ntafos in 1984 countered this by describing the
merits of using random input testing.

The term “fuzzing” was first coined by Miller in 1988 as part of a class project determining why line noise over a “fuzzy” modem connection would crash many UNIX utilities. This research was extended by Miller et al. to investigate the causes of crashes on a wide range of UNIX systems. While they focus on userspace utilities rather than kernel interfaces, many of the bugs they find (including NULL pointer dereferences and lack of bounds checking on arrays) are the same as those found by us with perf_fuzzer 25 years later.

Miller et al. revisited tool fuzzing in 1995 and found that they could still crash over 40% of common system utilities on UNIX and Linux systems. Forrester and Miller extended the work to look at Windows NT and Miller, Cooksey and Moore looked at Mac OSX userspace programs and found similar userspace error rates to those on UNIX. Most of these investigations look at userspace utilities; our work concentrates on operating system kernel interfaces.

Operating systems have many potential interfaces exposed to users that can harbor bugs. Carrette’s CrashMe project attempts to crash the operating system by fuzzing the instruction stream. Unlike our work, this does not target system calls directly, but the entire operating system and underlying hardware in the face of random processor instructions. Medonça and Neves fuzz at the device driver level by externally sending malicious inputs to wireless networking hardware. Cadar et al. use an analysis tool that examines executables and generates inputs based on program flow; they apply this to finding crashing bugs in the Linux filesystem code with malicious filesystem images. Another interface open for bugs in modern systems is the virtual machine interface.

Koopman et al. look at the robustness of five different operating systems (Mach, HP-UX, QNX, LynxOS and Stratus FTX) by injecting random data at the operating system interface, focusing on seven commonly used system calls: read(), write(), open(), close(), fstat(), stat(), and select(). On four of the five systems bugs severe enough to require a restart were found. Our work is similar to this, but involves focusing on a single system call on the Linux operating system.

Existing Linux system call fuzzers such as Jones’ Trinity and Ormandy’s iknowthis test the majority of available system calls with varied parameters. They currently do not focus on one system call, and only have limited support for using system call dependency information to chain together related system calls the way that perf_fuzzer can. perf_fuzzer shares some code with Trinity; this will be described in more detail in Section.

Vyuok’s syzkaller fuzzes the kernel as well. It depends on custom written templates that describe the system calls, but adds coverage-based support that instruments the kernel to automatically detect when a fuzzed access has caused issues. While this is much more powerful than plain random testing, it still misses some of the bugs found by perf_fuzzer, as we use detailed knowledge of what makes a valid sequence of perf_event related system calls, some-thing that is hard for a fully automated fuzzer to work out experimentally.

Oehlert describes fuzzing a Windows terminal program and provides some useful definitions. He notes that the most critical bugs found with fuzzers are those that cross a trust boundary (user to kernel or network to local). He differentiates between two techniques for creating inputs: data generation and data mutation. The former is when inputs are randomly chosen based on a specification while the latter takes known working inputs and modifies them slightly. A related distinction is intelligent and unintelligent fuzzers: the former knows what valid input looks like and attempts to mimic it, the latter just generates inputs randomly. By these definitions, our perf_fuzzer does data generation while attempting to be an intelligent fuzzer.

3. MOTIVATION

Despite the best efforts of the maintainers, bugs are continually found in operating systems such as Linux. This work concentrates on finding bugs in the Linux perf_event performance monitoring subsystem which was introduced in the 2.6.31 kernel in 2009. The perf_event interface is more complex than the competing interfaces it replaced, and it has only grown more complicated as it has accumulated features since its introduction.

The perf_event interface was chosen as the target of domain-specific fuzzing due to our ongoing frustration with finding bugs in the interface and hoping to automate the process (rather than building a reactionary bug test suite such as perf_event_test). We help develop the PAPI performance library which is widely used by the high-performance computing community. PAPI tends to exercise a different subset of functionality than the more commonly used perf command-line utility-distributed with the Linux kernel source. Since most kernel developers restrict their perf_event usage to perf, any functionality not exercised by that tool can break without being noticed. Work on PAPI has turned up numerous kernel bugs, as seen in Table 1. These issues were all found by programs trying to exercise normal, expected functionality of the interface. This hinted that even more bugs would be exposed by more methodical testing (such as fuzzing) and indeed that is what we find.

The Trinity fuzzer (described in more detail in Section) added support for perf_event_open() soon after the system call was introduced. Trinity initially had limited support for the call, making it extremely unlikely that valid or near-valid events would be generated. We contributed slightly better support in November 2011 as an ongoing part of research into the interface. Not much came of this until April 2013 when Rantala found a bug using Trinity where the 64-bit attr.config value was being copied to a 32-bit integer before being sanity checked. This bug meant that the high 32-bits could be controlled by the user, and eventually it was discovered that this could be exploited by a local user to get root privileges (CVE-2013-2094). More worrisome, the kernel code change that introduced this bug happened in 2010 and was possibly being exploited soon after, but it took 3 years for the bug to be found and fixed.

The publicity surrounding this security breach renewed our
Table 1: Linux kernel perf_event security bugs from 2009-2013 found without fuzzers.

<table>
<thead>
<tr>
<th>Type</th>
<th>CVE</th>
<th>Fixed (version/git commit)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>root exploit</td>
<td>CVE-2009-3234</td>
<td>2.6.32 b36c62c365a586744</td>
<td>buffer overflow</td>
</tr>
<tr>
<td>crash</td>
<td>CVE-2010-4169</td>
<td>2.6.37 6b3617384b119469</td>
<td>improper mmap hook</td>
</tr>
<tr>
<td>crash</td>
<td></td>
<td>2.6.39 ab71f6e68297d14</td>
<td>task context scheduling</td>
</tr>
<tr>
<td>memleak</td>
<td></td>
<td>2.6.39 38b4355b1e636b42e</td>
<td>inherited events leak memory</td>
</tr>
<tr>
<td>crash</td>
<td>CVE-2011-2521</td>
<td>2.6.39 fd66c2100e2539e</td>
<td>x86 msr registers wrong</td>
</tr>
<tr>
<td>DoS</td>
<td>CVE-2011-2918</td>
<td>3.1.1 a8b0ca17b80e92fa</td>
<td>software event overflow</td>
</tr>
<tr>
<td>crash</td>
<td>CVE-2011-2918</td>
<td>3.1.1 a8b0ca17b80e92fa</td>
<td>software event overflow</td>
</tr>
<tr>
<td>crash</td>
<td></td>
<td>3.5.1 9c5da09d266ca993</td>
<td>cgrou reference counting</td>
</tr>
<tr>
<td>crash</td>
<td>CVE-2013-2146</td>
<td>3.9.1 f1923820c447e986</td>
<td>offcore mask allows writing reserved bit</td>
</tr>
<tr>
<td>crash</td>
<td></td>
<td>3.9.1 1d0d8639c063ca6f</td>
<td>pebs/bits state after suspend/resume</td>
</tr>
</tbody>
</table>

interest in perf_event fuzzing. We sent enhanced patches to Trinity to bring it in line with modern kernels, but also started development of the perf_fuzzer in May 2013 to go above and beyond the coverage offered by Trinity.

4. BACKGROUND AND IMPLEMENTATION

4.1 The perf_event Interface

The perf_event performance monitoring subsystem has a complex interface that is not completely exercised by a naïve fuzzer. A full description of the interface can be found in the perf_event_open.2 manpage [30]. The perf_event_open() interface is complex enough that it has the longest manual page of any system call, longer even than the elaborate ptrace() system call.

The prototype for the system call looks like this:

```c
int perf_event_open(struct perf_event_attr *attr, pid_t pid, int cpu, int group_fd, unsigned long flags);
```

It takes five input arguments:

- `attr` is a complicated structure describing the event to be created with 40 inter-related fields (see Appendix A for more details),
- `pid` specifies which process id to monitor (0 indicating current, -1 indicating all),
- `cpu` specifies which CPU core to monitor (-1 indicating all),
- `group_fd` allows an event to join a group leader, creating a group of events that can be read simultaneously,
- `flags` allows setting various optional event flags.

There are two common ways of using perf_event: one is monitoring any program belonging to a user (anyone can do this by default), the other is system-wide measurement (which generally requires root permissions to avoid leaking sensitive information between users).

Opening an event with perf_event_open() is only a small part of the perf_event experience. Many bugs that are found do not happen solely at open, but also depend on interactions with other calls. Various other kernel interfaces interact with perf_event:

- `prctl()` (process control) can be used to start and stop all events in a process,
- `ioctl()` is used to start, stop, and otherwise get information about events,
- `read()` returns the current values of counters and some additional information,
- `mmap()` can map pages that provide event info as well as a circular ring buffer where the kernel places sampled event information,
- `poll()` can wait for overflow or buffer-full signals,
- and, various files under /proc and /sys provide extra event information and configuration settings.

The perf_event implementation involves low-level code scattered throughout the kernel, making the interface complex to debug. Hardware events are generally programmed by writing to CPU model specific registers (MSRs on x86). Hardware events can overflow, triggering non-maskable (NMI) interrupts. Software events (counts of kernel maintained values such as context-switches and interrupt counts) require placing perf_event code in time critical kernel functions. The perf_event interface has also grown to include the hardware breakpoint interface and has major connections to the ftrace system tracing interface. In addition support has been added to support running Berkeley Packet Filter (BPF) programs in the kernel in conjunction with events, further increasing the potential sources of bugs.

4.2 The Trinity Fuzzer

Jones introduced the Trinity fuzzer [12, 13], first as scrashme in 2006, and then renamed Trinity in 2010. The tool is designed to methodically check all of the Linux system calls looking for bugs that affect the kernel. Trinity excels at creating “interesting” inputs: rather than always passing purely random values into the kernel, it picks values that are valid, close to valid, or known boundary or corner cases. For string cases it generates not only normal ASCII strings but pathological cases with lots of nulls or weird Unicode values. It
also creates resources commonly used as inputs to syscalls, such as pre-initialized file descriptors and chunks of allocated memory.

The following annotations can be provided for system call inputs:

- ARG_RANDOM_LONG – random long integer, with special code to mix in “interesting” values,
- ARG_FD – random pre-defined file descriptor from a list containing various interesting files in /dev, /proc, /sys, perf_event, pipes, network sockets, etc.,
- ARG_LEN – random size of a variable,
- ARG_ADDRESS / ARG_NON_NULL_ADDRESS – random memory address,
- ARG_MODE_T – random access mode,
- ARG_PID – random process ID,
- ARG_RANGE – random value from a provided range,
- ARG_LIST / ARG_OP – random bit masks composed by or-ing values from a provided list,
- ARG_RANDPAGE – a page full of random values,
- ARG_CPU – random cpu,
- ARG_PATHNAME – random path name,
- ARG_IOVEC / ARG_IOVECLEN – random iovec,
- ARG_SOCKADDR / ARG_SOCKADDR_LEN – random socket,
- or, ARG_MMAP – random mmap() mapping.

An additional sanitise routine can be provided which cleans up the randomly selected parameters to make them more likely to be valid.

When started, Trinity initializes various structures, such as randomized memory pages and file descriptor tables. A number of children are created which do the actual fuzzing. A watchdog process is also created that makes sure the children are making forward progress and restarts them if they die. The children record their progress to a log file, syncing to avoid losing information in a crash. Ideally the fuzzer can run forever without incident, but usually at some point some sort of kernel message, panic, or crash will happen which then needs to be identified and reported.

4.3 The perf_fuzzer
Trinity does a remarkable job of finding bugs, but it currently runs system calls mostly independently. An interface like perf_event often has bugs that involve various system calls interacting in a complex set of ways that are hard to describe with the current Trinity infrastructure. Figure 1 shows at a high level how a generic syscall fuzzer differs from a targeted fuzzer such as perf_fuzzer.

4.3.1 Implementation
The perf_fuzzer re-uses the syscalls/perf_event_open.c fuzzing routines provided by Trinity. Sharing code between the two projects avoids duplicated work and ensures that any improvements in one project are included in the other. The perf_fuzzer does not directly use any Trinity interfaces besides the syscall_perf_event_open.sanitise() call that initializes and sets up the arguments for the system call.

At startup the perf_fuzzer parses the command line. It seeds the random number generator, either based on the time, or else via a value passed by the user (to enable re-running with same initial start conditions). This value is also printed and written to disk to ease reproduction of a run. The process id is logged so that during replay any invocations using the previous process id are re-mapped to the current one. Various structures are initialized, including calling the Trinity syscall_perf_event_open.init() routine and creation of a Trinity-compatible “page_rand”.

Next the signal handlers are initialized. These can be a source of errors as the more widely used perf utility does not use signal handlers (it uses poll() to detect overflows). The perf_fuzzer sets up counter overflows to trigger SIGRT signals (as PAPI does) because they queue and avoid losing signals when a system is busy. Eventually the queues can fill and the kernel handles this by sending SIGIO: we set up handlers for both SIGRT and SIGIO. The SIGRT handler disables the event causing the signal, reads event values and then restarts the event. If the SIGIO handler is triggered it means we are stuck in a tight overflow storm and not making forward progress, so it attempts to close the event causing the issues (this is difficult, especially if the event was created in another thread before forking). An additional SIGQUIT handler is set up that will dump the current open event state so a user can monitor the current status of the fuzzing.

The main perf_fuzzer event loop is then entered, which loops forever randomly selecting one of the following tasks. These tasks have been arbitrarily chosen based on knowledge of the interface and the tools that typically use it.

- Open a Random Event
Repeatedly run perf_event_open() with random parameters until it successfully creates an event. It reuses the Trinity syscall sanitise code, which:
  1. clears the fields,
  2. randomly sets cpu to -1 (any) or else a valid CPU,
  3. sets group_leader to -1 (I’m a leader) or a random other id,
  4. sets flags to one of four valid values or else completely random,
  5. sets pid to either the current pid, 0 (which means current), -1 (all), or a random pid,
  6. then it sets up the attr structure to one of 3 choices: a mostly valid counting event, a mostly valid sampling event, or a completely random event.

The following are the possible attr_type field settings; perf_fuzzer tries to exercise them all. It also chooses
appropriate random `config` values appropriate for the `type` selected:

- HARDWARE – the kernel defines various pre-defined “generic” events
- HW_CACHE – these are also pre-defined events but with a more complex encoding
- RAW – these are the raw values passed directly to the underlying CPU and vary based on architecture and processor model. perf_fuzzer does not currently make intelligent picks here
- SOFTWARE – the kernel defines various events
- BREAKPOINT – hardware breakpoints, we try to pick mostly valid size and address fields as well as read, write, or execute settings
- TRACEPOINT – possible values can be read from debugfs but it is rare that this is mounted; the values are usually small so we preferentially choose a low random integer
- SYFS – some common generic events are exported via the sysfs filesystem in a complex series of files under `/sys/bus/event_source/devices/`. At startup these values are parsed and randomly selected
- RANDOM – other performance measurement units (PMUs) can be available; they are dynamically assigned values above the last pre-defined kernel type.

In addition there are a few other fields that need to be set for a valid event. There is `attr.size` which is used for versioning and backward compatibility and it needs to be one of a few possible values. There are also various boolean flags that are chosen randomly. Once an event is opened, the fuzzer randomly decides whether to enable an associated `mmap()` ring buffer page or overflow signal handler.

Despite the fuzzer’s advanced event creation knowledge, a large percentage of events (usually more than 99.8%) fail to open. To ensure useful behavior the fuzzer loops creating events until a valid one is generated.

- **Close a Random Event**
  Randomly choose an active event and closes it. If the event had mmap’d any memory, unmap it. Originally the code randomly chose to not unmap, but this ended up leaking enough memory to cause problems without finding any kernel bugs.

- **Ioctl a Random Event**
  Randomly choose an open event and performs an `ioctl()` on it. `ioctl()`s by definition are very specific to the interface being controlled; ioctl fuzzing is hard to do generically (a limitation of Trinity and other fuzzers).

  perf_fuzzer has special knowledge of perf_event related `ioctl`s. It generates one of the following:

  - `PERF_EVENT_IOC_ENABLE` – enable an event
  - `PERF_EVENT_IOC_DISABLE` – disable an event
  - `PERF_EVENT_IOC_REFRESH` – restart an event for a certain number of overflows
  - `PERF_EVENT_IOC_RESET` – reset the event counts
  - `PERF_EVENT_IOC_PERIOD` – set the overflow period
  - `PERF_EVENT_IOC_SET_OUTPUT` – redirect event notifications to another fd

Figure 1: Comparison of a generic syscall fuzzer (such as Trinity) with a targeted fuzzer (such as perf_fuzzer).
The perf_fuzzer also assigns a random argument, a split between 0, PERF_IOC_FLAG_GROUP (the only defined flag which in theory causes the ioctl to affect all events in a group but which was broken for many kernel versions), or a random value.

- **Prctl the Process**
  Randomly execute the prctl() process control routine with PR_TASK_PERF_EVENTS_ENABLE or DISABLE which enables or disables all events in a process.

- **Read Random Event**
  Randomly read from an active event file descriptor. The size is randomly picked to be either the expected size based on the event creation flag, or a completely random size.

- **Write Random Event**
  Randomly write to an active event file descriptor. This is currently unsupported by the perf_event interface so likely never will trigger any bugs, but was included for completeness.

- **Access a Random File**
  Randomly read or write from a perf_event related file in the proc or sys filesystems. For example:
  - /proc/sys/kernel/perf_event_paranoid
  - /proc/sys/kernel/perf_event_max_sample_rate
  - /sys/kernel/perf_event_mlock_kb
  - /sys/bus/event_source/devices/

Most of these need root access so it is not likely to trigger bugs unless the fuzzer is run as root.

- **Fork the Process**
  Call fork() in an attempt to find any threading related bugs. Open file descriptors and signal handlers are inherited by the child, so the potential for bugs exists. perf_fuzzer currently has a simple implementation: it will only fork one child, and only if none has already been forked. If a child exists, then it is killed. The child simply sits in a busy loop. Even this simple behavior causes a lot of bugs, more complex child behavior may be added in the future.

- **Poll an Event**
  The perf tool uses the poll() system call when measuring overflow events. When the mmap’d buffer crosses a specified threshold the poll returns and data can be read. The perf_fuzzer picks a random number of active events and then polls on them. Right now a fairly short timeout is used as not to hold up the fuzzing process.

- **Corrupt the mmap Page**
  Sampled events mmap() a circular ring buffer from the kernel. This is writable by the user so that a tail pointer can be adjusted (so the kernel can avoid overwriting values that have not been read yet). The perf_fuzzer writes random values into the mmap page to try to trigger bugs.

- **Run a Million Instructions**
  Last in the list is an assembly language routine that runs for a million instructions without running any syscalls.

  Each time through the event loop the overflow refresh threshold is randomly updated (this would make more sense in the refresh signal handler, but that is not possible as rand() is not signal safe).

  Every 10,000 iterations a status message is printed; more details can be found in the Appendix.

### 4.3.2 Reproducibility

One highly desirable trait of a fuzzer is that it has reproducible results: given the same random seed the same exact values are generated by the fuzzer. This can greatly ease debugging of problems, and is useful for creating regression tests to verify if a particular bug has been fixed.

The perf_fuzzer has been carefully written to be as reproducible as possible, although full determinism is not always possible when measuring performance events because outside factors (such as hardware interrupts, kernel interactions, and other system activity) can vary from run to run. Event availability can vary between kernel versions and processor types, further reducing the possibility of deterministic results.

To ease reproducibility, a header is generated which includes enough information to recreate a fuzzing run. This makes it easy to include this state into bug reports and allows more easily recreating test conditions that cause failures. The header includes the version of perf_fuzzer, the Linux version and architecture, and the processor type. Also included is the random number seed, which allows replicating the random number generation exactly. Some kernel settings are also saved, such as the /proc/sys/kernel/perf_event_max_sample_rate value controls the maximum event sample rate. If this value differs from the original run then some events may fail because they set the sample rate too high. This is a particularly tricky value, as the kernel will automatically adjust this downward (outside of user control) if it thinks interrupts are happening too quickly. Another kernel value is /proc/sys/kernel/perf_event_paranoid. This allows the system administrator to allow access to some events (such as system-wide events) that are disabled by default for normal users for security reasons. If this value differs from the default then some events that would normally fail will instead open without error.

One last issue with reproducibility is whether failed system call attempts need to be recorded. In general only successful calls should affect kernel state, but it is conceivable that

```plaintext
- PERF_EVENT_IOC_SET_FILTER – attach a ftrace filter to the event. perf_fuzzer does not do as much with this as it could, as the filters generally require root and debugfs to be mounted
- PERF_EVENT_IOC_ID – return the unique event ID generated by the kernel
- generate a completely random ioctl (likely invalid).
```
an out of range value could cause a problem before the call fails. This can be a problem with logging, as the perf_fuzzer generates an order of magnitude more failed calls than successful ones. By default failed calls are not logged, but they can optionally be enabled for enhanced debugging.

4.3.3 Isolating and Reporting Bugs

To use the fuzzer, simply compile it, run it, and watch the system logs for error reports. For best results use a serial console to a separate machine; in the event of a crash a machine can lock up before logs and messages can be written to local disk.

In simple cases a panic will be generated that can be debugged by the user or sent to the linux-kernel list for analysis. Often the issue is complicated, and it can take time to isolate the bug and generate a useful bug report.

To make it easier to reproduce bugs, it is often useful to have a number of short runs (stopping after 50,000 events or some other small number) rather than one long fuzzing run. Replaying and finding a bug that happens after a few seconds is a lot easier than trying to reproduce one that occurs after a week of runtime. The fuzzer also has options to limit which particular system calls to fuzz, allowing one to narrow down the scope of the fuzzing.

4.3.4 Logging and Replay

perf_fuzzer has a logging mode that can be enabled. An ASCII text file is generated: for each action a letter indicating the action type is printed followed by a list of the parameters needed to replay the action.

Logs quickly get large and the entire file contents can be important. Bugs are often not simply caused by the last perf_event_open() call, but by a long chain of related actions scattered throughout the log. Determining the last action that causes a lockup can be difficult as crashes can happen quickly enough that key values are not logged to disk. Even running sync() before logging is not always enough to capture the value (and that slows the fuzzing process). The behavior of the fuzzer is usually deterministic enough that multiple runs with the same random seed usually get to the same place, so a special trigger can be inserted in the code to pause just before the last problem causing action.

Replaying a log and generating the same system call trace is a fragile process. Iterating through the log file and generating the system calls therein is often enough to reproduce bugs, but regenerating exact behavior takes special care.

An exact replay requires generating file descriptor numbers that match those from the original run. When logging is enabled, an extra log file descriptor is created that would not be there in a non-logging run. To adjust for this we allocate a dummy log file, even if not logging, so that the file descriptors match up.

The exact number of open(), close(), mmap(), and read() system calls can subtly affect replay. The perf_fuzzer does a number of these on setup, to read and print the system information as well as scan the /sys directory for event names. To ensure the same total system call count we make the the replay code run through the same init code as the actual fuzzer.

The memory addresses returned by system calls can change, especially for anonymous mmap() calls. To get consistent memory address locations the address space randomization feature of the Linux kernel needs to be disabled (this can be done by setting /proc/sys/kernel/randomize_va_space to 0). In addition any mmap()’s done by the fuzzer (usually these involve ones mapping buffers for console output) must also be matched in the replay.

When conducting tests involving the fork() system call, identical thread interleaving is important. If a killed child thread takes a different amount of time to deallocate its events, an attempted event opening in the parent thread can fail if a resource limit is hit. To enable deterministic fork behavior the fuzzer and the replay code should both include waitpid() calls to make sure a child dies completely before continuing.

4.3.5 Tools

We have developed additional tools that can help analyze the log files:

- **replay_log** takes a log and replays all the events. Due to the nature of perf_events (many are non-deterministic) this does not always generate the exact same execution, especially with things like signal handlers.

  Once you have a log that causes a bug/crash and replay_log reproduces it, you can isolate the problem. One way is a binary search (or “bisect” in kernel terms). Currently this is done manually. This could probably be automated, but the process often requires manual intervention anyway to reboot after each crash.

- **filter_log** can filter logs by action type to reduce the size by eliminating actions not likely to cause the bug (writes, opens, forks, etc).

- **active_events** analyzes a log and prints the active events at the time of the end of the log.

- **log_to_code** takes a log and converts to a valid C program that will replay the log. This is useful for creating small reproducible test cases, and is also good at turning the long string of values in a line of the log into something human readable.

5. RESULTS

Table 2 summarizes the major perf_event bugs that have been found (and subsequently fixed) by Trinity and perf_fuzzer from April 2013 through February 2019. Over twenty major bugs have been found, which is more than those found by more traditional methods over the preceding four years as shown earlier in Table 1.

5.1 Critical Bugs Found

perf_fuzzer triggers a wide variety of bugs; not all of them are dangerous or security issues. What follows is a summary of the types of issues we have found thus far.
Table 2: Linux perf event security bugs found by fuzzers starting from April 2013. (T=Trinity, P=perf_fuzzer, H=honggfuzz [26], S=syzkaller [29])

<table>
<thead>
<tr>
<th>Type</th>
<th>CVE</th>
<th>Fixed in Linux</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>root exploit</td>
<td>CVE-2013-2094</td>
<td>3.9 8176cced70dbb5e5d</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>3.10 9b5d40dc93c93d4d</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>3.10 26cb53e111e01047</td>
</tr>
<tr>
<td>P</td>
<td>panic</td>
<td>-</td>
<td>3.11 d99966357b14e356</td>
</tr>
<tr>
<td>P</td>
<td>root exploit</td>
<td>CVE-2013-4254</td>
<td>3.11 39eb3184ea1a3a2</td>
</tr>
<tr>
<td>P</td>
<td>panic</td>
<td>-</td>
<td>3.11 868918a8a1f06</td>
</tr>
<tr>
<td>P</td>
<td>panic</td>
<td>-</td>
<td>3.11 ee532a08a435050</td>
</tr>
<tr>
<td>P</td>
<td>panic</td>
<td>-</td>
<td>3.13 9eb29825b81dca9</td>
</tr>
<tr>
<td>P/T</td>
<td>crash</td>
<td>CVE-2013-2930</td>
<td>3.13 12ac003d54e15207</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>3.14 0a99698c61b06</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>3.15 46ece097a6be753</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>3.15 4b1e726a3308c2</td>
</tr>
<tr>
<td>P</td>
<td>reboot</td>
<td>-</td>
<td>3.17 3577a77a202c4813</td>
</tr>
<tr>
<td>P</td>
<td>exploit</td>
<td>CVE-2015-9004</td>
<td>3.19 9fc81d87420d3f1</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>3.19 98c0086d8852655</td>
</tr>
<tr>
<td>P</td>
<td>exploit</td>
<td>CVE-2015-3004</td>
<td>3.19 9fc81d87420d3f1</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>3.19 98c0086d8852655</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>CVE-2015-8955</td>
<td>4.1 8fff105e135a44f</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>4.1 26e61e89b1fe87</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>4.1 2a839282e435924</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>4.1 26e61e89b1fe87</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>4.1 2a839282e435924</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>4.1 2a839282e435924</td>
</tr>
<tr>
<td>P</td>
<td>memleak</td>
<td>-</td>
<td>4.1 2a839282e435924</td>
</tr>
<tr>
<td>H</td>
<td>memleak</td>
<td>-</td>
<td>4.1 2a839282e435924</td>
</tr>
<tr>
<td>P</td>
<td>exploit</td>
<td>CVE-2017-6001</td>
<td>4.10 321027c1fe77f892</td>
</tr>
<tr>
<td>S</td>
<td>bug</td>
<td>-</td>
<td>4.11 502a8389a4e09</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>4.15 99a9dc98ba52267c</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>4.19 7ccc4fe5ff93a13</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>4.19 6e2713ed7e702f</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>4.19 8e5b176f36348f1</td>
</tr>
<tr>
<td>P</td>
<td>BUG</td>
<td>-</td>
<td>4.19 7ccc4fe5ff93a13</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>4.19 6eb204f360f25</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>-</td>
<td>4.20 9df20aa96a32626</td>
</tr>
</tbody>
</table>

Table 3: Linux perf_event WARNING and BUG assertions found by fuzzers (T=Trinity, P=perf_fuzzer, Z=trinity run by 0-day tester)

<table>
<thead>
<tr>
<th>Type</th>
<th>CVE</th>
<th>Fixed in Linux</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>WARNING</td>
<td>3.11 734df5ab549a444</td>
<td>WARNING: at kernel/events/core.c:2122</td>
</tr>
<tr>
<td>P</td>
<td>WARNING</td>
<td>3.14 26e61e8939b1e87</td>
<td>WARNING: at arch/x86/kernel/cpus/perf.c:1076</td>
</tr>
<tr>
<td>T,Z</td>
<td>BUG</td>
<td>3.17 next caught early</td>
<td>BUG: unable to handle kernel NULL pointer</td>
</tr>
<tr>
<td>P</td>
<td>WARNING</td>
<td>3.19 96e1d87420d3f1</td>
<td>WARNING: Can’t find any breakpoint slot</td>
</tr>
<tr>
<td>P</td>
<td>BUG</td>
<td>3.19 a9f1560e762d0493</td>
<td>Bug: uncore_assign_events()</td>
</tr>
<tr>
<td>T</td>
<td>WARNING</td>
<td>4.0 321027c1fe77f892</td>
<td>WARNING: add_event_to_ctx()</td>
</tr>
<tr>
<td>P</td>
<td>WARNING</td>
<td>4.1 26e61e8939b1e87</td>
<td>WARNING: add_event_to_ctx()</td>
</tr>
<tr>
<td>P</td>
<td>WARNING</td>
<td>4.2 b4875b67e6813</td>
<td>WARNING: add_event_to_ctx()</td>
</tr>
<tr>
<td>P</td>
<td>WARNING</td>
<td>4.2 93472affd024d76</td>
<td>WARNING: Fix active_events imbalance</td>
</tr>
<tr>
<td>P</td>
<td>BUG</td>
<td>4.9 c49336e8a6bce15</td>
<td>BUG: KASAN: slab-out-of-bounds</td>
</tr>
<tr>
<td>P</td>
<td>WARNING</td>
<td>4.9 e552a8389a409</td>
<td>Fix use-after-free in perf_release()</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>4.9 2b2226676704f29</td>
<td>Fixing buffer recycling</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>4.15 99a9dc98ba52267c</td>
<td>BTS causes crash with KPTI meltdown fixes</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>4.19 7ccc4fe5ff93a13</td>
<td>WARNING: add_event_to_ctx()</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>4.20 472de491dc3365</td>
<td>BTS crash, KPTI meltdown fixes</td>
</tr>
<tr>
<td>P</td>
<td>crash</td>
<td>4.20 81ce31c4c4d782</td>
<td>BTS crash, KPTI meltdown fixes</td>
</tr>
</tbody>
</table>
Table 4: Linux perf_event correctness bugs found while using fuzzer. (T=Trinity, P=perf_fuzzer)

<table>
<thead>
<tr>
<th>Which</th>
<th>Type</th>
<th>Fixed in Linux</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Aliasing</td>
<td>3.13</td>
<td>16024e2dd4a7d8a87</td>
</tr>
<tr>
<td>P</td>
<td>Correctness</td>
<td>3.15</td>
<td>ftrace config value 64-bit but only lower 32 checked</td>
</tr>
<tr>
<td>P</td>
<td>Correctness</td>
<td>3.16</td>
<td>sample_period unsigned cast to signed</td>
</tr>
<tr>
<td>P</td>
<td>Correctness</td>
<td>4.11</td>
<td>flags value 64-bit but only lower 32 checked</td>
</tr>
<tr>
<td>P</td>
<td>Correctness</td>
<td>4.11</td>
<td>Fix perf_cpu_time_max_percent check</td>
</tr>
<tr>
<td>?</td>
<td>Wrong Resource</td>
<td>4.15</td>
<td>RAPL readings using wrong MSRs</td>
</tr>
</tbody>
</table>

Table 5: Linux perf_event CVE bugs that look like the type found with fuzzers, but I have not been able to confirm that they were found that way.

<table>
<thead>
<tr>
<th>Which</th>
<th>Type</th>
<th>CVE</th>
<th>Fixed in Linux</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>exploit</td>
<td>CVE-2018-1000199</td>
<td>4.16</td>
<td>f67b15037a7a50 modify_user_hw_breakpoint()</td>
</tr>
<tr>
<td>?</td>
<td>exploit</td>
<td>CVE-2017-6001</td>
<td>4.10</td>
<td>321027c1fe77f8 Race condition in kernel/events/core.c</td>
</tr>
<tr>
<td>?</td>
<td>exploit</td>
<td>CVE-2016-6787</td>
<td>4.0</td>
<td>f63a8d2aa5812af kernel/events/core.c mismanages locks (A-31095224)</td>
</tr>
<tr>
<td>?</td>
<td>exploit</td>
<td>CVE-2016-6786</td>
<td>4.0</td>
<td>f63a8d2aa5812af kernel/events/core.c mismanages locks (A-30955111)</td>
</tr>
<tr>
<td>?</td>
<td>exploit</td>
<td>CVE-2016-3843</td>
<td>?</td>
<td>? Elevation of privilege, Qualcomm (A-28086229)</td>
</tr>
<tr>
<td>?</td>
<td>exploit</td>
<td>CVE-2016-3768</td>
<td>?</td>
<td>? Elevation of privilege, Qualcomm (A-28172137)</td>
</tr>
<tr>
<td>?</td>
<td>exploit</td>
<td>CVE-2016-0843</td>
<td>?</td>
<td>? Elevation of privilege, Qualcomm (A-25801197)</td>
</tr>
<tr>
<td>?</td>
<td>exploit</td>
<td>CVE-2016-0819</td>
<td>?</td>
<td>? Elevation of privilege, Qualcomm (A-25364034)</td>
</tr>
<tr>
<td>?</td>
<td>exploit</td>
<td>CVE-2016-0906</td>
<td>?</td>
<td>? Elevation of privilege, Qualcomm (A-25773204)</td>
</tr>
<tr>
<td>no</td>
<td>DoS</td>
<td>CVE-2015-5963</td>
<td>4.10</td>
<td>12ca6ad2ec3a999 race on CPU unplug (A-30992077)</td>
</tr>
<tr>
<td>no</td>
<td>DoS</td>
<td>CVE-2015-6526</td>
<td>4.1</td>
<td>9a5cbce4212f283 powerpc: Cap 64-bit userpace backtraces</td>
</tr>
<tr>
<td>no</td>
<td>DoS</td>
<td>CVE-2014-7826</td>
<td>3.18</td>
<td>086ba77af6d1b00e NR_syscall out of range on ARM</td>
</tr>
<tr>
<td>no</td>
<td>DoS</td>
<td>CVE-2014-7825</td>
<td>3.18</td>
<td>086ba77af6d1b00e NR_syscall out of range on ARM</td>
</tr>
</tbody>
</table>

5.1.1 Crash / Hang / Panic / Denial of Service

The most annoying type of bug found is one that completely crashes the computer. Tracking down this type of bug is difficult as logging and debugging information are often lost.

A related issue is where a bug manages to cause a process to become stuck and hang one of the processor cores. In this case often the operating system watchdog will kick in and give some information on the problem, or otherwise the Linux “ALT-SYSRQ” stack backtrace functionality can be used to debug the problem.

Sometimes the error will be one where an invalid memory access is triggered in the kernel; this will cause a kernel panic. This type of bug is often easier to isolate due to the debug information provided by the panic message.

These bugs have security implications; at the very least they are “Denial of Service” (DoS) attacks. Even in cases where the operating system does not crash outright, often the system will be left in an unusable or fragile state that needs rebooting. These bugs can often be triggered by a regular user to make the system unavailable. Despite this, reports of this nature are treated with fairly low urgency by the perf_event developers unless a small triggering case can be created.

5.1.2 Hang Example

An example of this type of bug is the “perf/ftrace wrong permissions check” bug fixed in the 3.13 kernel. The ftrace infrastructure allows creating perf_event events that trigger at various predefined code locations in the kernel. The fuzzer created an event that caused an overflow on every function entry; if set up to overflow, then the overflow handler will trigger this event, which can recursively cause another overflow which triggers another event, etc., causing the kernel to get trapped in an endless loop. The machine will become unresponsive at this point, although the watchdog might eventually kick in and display a “kernel is stuck” message.

Once reported this bug was not really fixed; instead the perf_event permissions were changed so that non-root users cannot create kernel function trace events. This was the original intention of the code, but due to the (somewhat confusing) nature of the internal perf_event permissions checking a comparison was coded wrong.

5.1.3 Local Root Exploit

Sometimes a bug that only looks like a crash or panic can turn out to have far greater security implications. If a bug lets user-supplied values get written into unexpected parts of kernel memory, eventually a clever user will be able to figure out how to use this to escalate their privileges and obtain root access.

The perf_event vulnerability that prompted the design of perf_fuzzer was such an exploit. An improperly checked config value for a software event allowed a user to arbitrarily increment any memory location. It was possible to use this to redirect the undefined instruction interrupt vector to point to user-supplied code, which then can carry out the privilege escalation (Edge [7] describes this in more detail).

A different bug found by perf_fuzzer is the “ARM event validity” bug. The ARM validate_event() function called armpmu->get_event_idx() on the group leader for an event.
However if the group leader was not an armpmu type, then the function pointer called was just whatever arbitrary value happened to be at the memory offset past the end of the structure. If you were unlucky, this arbitrary value was a valid user address, and for a short window of time in the 3.11-rc cycle this value pointed to a value initialized to INT_MIN which is a valid user mappable address of 0x80000000. If a user mapped exploit code there, the kernel could escalate privileges, and we created demonstration code that did just this. Luckily this bug was found and fixed before it made it into a released kernel.

5.1.4 Warnings
Throughout the Linux kernel code are “warnings”: debug macros of the type WARN_ON used as asserts to catch corner cases the author of the code thinks are invalid but unlikely.

Fuzzers often trigger these messages. Sometimes the problem reported is real and can be fixed, sometimes it is a false positive and just silenced. It is still important to report these although such problems rarely cause crashes. A list of warnings triggered by perf_fuzzer can be seen in Table 2.

5.2 Other Bugs Found
There are perf_event bugs in the kernel that are not obviously security bugs, but just problems with the interface. Fuzzers are not designed to catch these bugs but sometimes they are noticed while tracking down more serious issues.

Table 4 shows bugs found where a 64-bit value was being range-checked with a 32-bit value. This is a common error when using preprocessor defined constants on 64-bit machines. The perf_fuzzer does not detect this type of bug; these are noticed manually in the log files when debugging other problems.

One example came up when debugging an ftrace problem. The fuzzer found a real bug, but the reproducible test case was odd. Only 32-bit config values were supposed to be supported by the interface, but the bug was triggering on an event that had 0x7fffffff as the top 32-bits of the 64-bit value. The ftrace had a bug where the value was being copied to a 32-bit value (which was truncated) to be checked for validity. This caused event aliasing where the top 32-bits were ignored. This was a correctness bug and was subsequently fixed, but was found only as a side effect of the actual fuzzing process.

5.3 Bugs Avoided
Now that the perf_fuzzer tool has become known in the kernel development community, it has started being used to catch bugs in patches before they are applied to the kernel tree. For example, the ARM perf_event developers encourage usage of perf_fuzzer during new patch submission.

5.4 Current Status
The current status of how successful the fuzzer is at crashing things quickly gets out of date as bugs are fixed. See Appendix C for an analysis from a few years back.

As of February 2019 and the 5.0 release the x86/Intel architecture can finally be fuzzed without quickly crashing. It took six years, but this is a huge help for debugging. We can finally track regressions instead of constantly finding existing bugs. Even though we were not quite there at the time, we were still able to quickly track down the BTS crashes caused by the KPTI fixes for the Intel Meltdown vulnerability because things crashed faster than normal.

Some platforms, such as ARM, have fairly good coverage upstream but the old back-ported kernels used in embedded devices such as cellphones tend to have bugs easily triggered by the perf_fuzzer. This is especially true as vendor-supplied (not upstream) perf event drivers are notoriously buggy. This has led to the perf_fuzzer being used by others to find many Android bugs, and possibly collecting bug bounties, although it has been surprisingly hard for us to track down the researchers involved to get full details.

6. FUTURE WORK
While the perf_fuzzer has already proved itself useful by finding a number of bugs in the Linux kernel, there are a number of future plans to improve the fuzzer in particular and the Linux kernel in general.

Improved Heuristics and Features
The subset of perf_event functionality explored by perf_fuzzer was based heavily on the areas exercised by the PAPI performance library. So many bugs were found with this first implementation that the addition of new features was stalled until the large backlog of existing problems were addressed. Recently most of the low-hanging bugs have been fixed, so we propose some new changes to improve code coverage:

- Testing more exotic ways of generating file descriptors, such as opened events being passed across an opened socket.
- Setting up breakpoints inside of perf_event data structures.
- Testing the perf_event cgroup (container) support. The perf_event interface supports special cgroup events, but the perf_fuzzer does not explicitly test this.
- More advanced coverage of multithreaded code. The current fork() fuzzing code is simplistic and does not test multiple children or errors caused by exec() of a new process.
- More intelligent raw hardware event choices. Currently the fuzzer picks raw hardware events completely at random. There are libraries that provide valid raw event values, such as libfpm4, that can be used to create more likely to be valid CPU events.
- Fuzzing the Berkeley Packet Filter (BPF) interface which can be used to enhance event collection.

Testing More Architectures
Another planned fuzzer improvement is widening the test coverage. Most of the fuzzing has been done on x86 systems (Core2, Haswell, and Skylake) as well as a few ARM systems (Cortex-A9 pandaboard and various Raspberry Pis). These
systems alone have found many bugs, but it would be good
to test other architectures, especially non-Intel systems, and
server systems that have more advanced performance units
with features such as Uncore, Offcore, and energy events.
The fuzzer can also be used to test emulated systems (such
as qemu) or the interfaces inside of virtual machines.

Code Coverage Awareness
When the fuzzer generates a new test case, it is currently un-
known whether this exercises a new path through the kernel
or is just a rehash of an already-tested path. Some fuzzers
(such as American Fuzzy Lop [33] and syzkaller [29]) are ca-
capable of using instrumentation to determine when new paths
are being explored. This is difficult to do with kernel code
without invoking massive slowdowns, but it might be possi-
ble to exploit the Branch Trace Store functionality available
on recent Intel processors to allow this kind of analysis.

Improved Determinism
One large impediment to finding bugs is the continued lack
of full determinism in the results, especially cross-platform.
The problem is that event generation repeats until a valid
event is chosen. The list of valid events (especially hardware
events) is tightly bound to the underlying CPU architecture,
and (to a more limited extent) the version of the operating
system kernel running. Therefore often buggy traces are
only reproducible on identical machines with similar kernel
versions. Changing this would require some major changes
to the underlying perf_fuzzer architecture and it might not
be possible to fully remove the determinism issues, even
though this would greatly ease reproducing bugs.

Enhancing Trinity (and other fuzzers)
Many of the techniques used with perf_fuzzer would be ap-
licable to testing other system calls on Linux. These can be
generalized and merged back into Trinity and other fuzzers
to allow better coverage without having to resort to special-
purpose niche fuzzers.

Improved Kernel Interface
The complex nature of the perf_event_open() system call
makes it a prime candidate for fuzzing. It is a large codebase,
not easily audited, and with many parameters that interact
in complex ways. One might wonder if is possible to design
a performance counter interface that would be less open for
these types of bugs.

The perfmon2 [9] interface was the leading candidate for an
official Linux counter interface before perf_event was merged.
In contrast to perf_event, it does many tasks (such as event
scheduling and event name mapping) in userspace instead
of the kernel. This reduces the size and attack surface of
in-kernel code. The interface has a much smaller number
of syscalls parameters, but does involve a much larger num-
ber of system calls (twelve). In this case it is unclear if the
interface would be more resistant to fuzzing or not.

Simpler interfaces exist, such as perfctr [24] (which does
most of its access via ioctl() and rdpmc()) and the sim-
ilar LiMiT [5] (which does most of its access via a simple
lprof_config() system call and rdpmc()). Again as much
as possible is done in userspace and the actual kernel inter-
face is limited to a simple interface to configure hardware
counters and fast reads of event values by special rdpmc
(read performance counter) CPU instructions. This type of
interface would seem at a first glance to be easier to analyze
(although ioctl() interfaces are unstructured and thus hard
to fuzz by general tools). The big drawback of these inter-
faces is the lack of features. The main benefit of perf_event
is the integration of all sources of performance information,
not just hardware performance counters, in one place. These
simpler interfaces do not allow access to the full range of
performance data available on a modern CPU.

Other proposals, such as LIKWID [28] bypass the kernel en-
tirely and depend on having raw access to the underlying
CPU registers. This has security issues of its own and is not
recommended for systems with hostile users.

Designing a kernel performance interface is a complex series
of tradeoffs, and it is unclear where the best mix of features,
complexity, and security lies. For Linux the path chosen was
perf_event, and for ABI stability reasons this is unlikely to
change. A major overhaul of the interface is unlikely, at best
if enough security issues are found the most likely outcome
is having the interface restricted to super-user access only.

7. CONCLUSION
The perf_fuzzer tool is a unique system-call specific fuzzing
tool that has found over twenty critical bugs in the Linux
kernel. These bugs found are over and above those found by
more generic fuzzers, showing that targeted domain knowl-
edge can find bugs that more generic fuzzers miss. Our
fuzzer also finds bugs that advanced, automatic code-coverage
fuzzers were not able to find.

Even though fuzzing is a well-known mature bug-finding
technology, we find that there is much room for improvement
in current fuzzers. Even incremental advances in fuzzing meth-
ology churn out large numbers of bug reports, show-
ing that operating system developers are not willing or able
to produce bug-free code with their current development
setup. Fuzzers provide valuable backup in catching errors,
such as a major system-lockup error with perf_event Branch
Trace (BTS) support introduced in the rush to get the fix
for the Intel Meltdown vulnerability out the door.

Kernel interfaces are not always designed with security in
mind. For complex interfaces like Linux perf_event fuzzers
are one of our best tools for ensuring operating system in-
tegrity. Operating system security is a difficult and thank-
less task but automated tools such as fuzzers that can find
bugs are a valuable tool in a security researcher’s arsenal.
8. AVAILABILITY
The perf_fuzzer tool is free software and is available from our website.

http://web.eece.maine.edu/~vweaver/projects/perf_events/fuzzer/

https://github.com/deater/perf_event_tests

9. REFERENCES
[23] T. Ormandy. iknowthis: i know this, it’s UNIX. http://code.google.com/p/iknowthis/
It takes five input arguments:

- **attr** is a complicated structure describing the event to be created with 40 inter-related fields.

```c
struct perf_event_attr {  
  __u32 type; /* Type of event */  
  __u32 size; /* Size of structure */  
  __u64 config; /* Type-specific config */  
  union {  
    __u64 sample_period; /* Sample period */  
    __u64 sample_freq; /* Sample frequency */  
  };  
  __u64 sample_type; /* Values in sample */  
  __u64 read_format; /* Values in read */  
  __u64 disabled; /* off by default */  
  __u64 read_value; /* Values in read */  
  __u64 read_start; /* Values in read */  
  __u64 read_mask; /* Values in read */  
  union {  
    __u64 config; /* Type-specific config */  
    __u64 size; /* Size of structure */  
    __u32 type; /* Type of event */  
  };  
};
```

- **pid** specifies which process id to monitor (0 indicating current, -1 indicating all),
- **cpu** specifies which CPU core to monitor (-1 indicating all),
- **group_fd** allows an event to join a group leader, creating a group of events that can be read simultaneously,
- **flags** allows setting various optional event flags.

### B. SAMPLE FUZZER OUTPUT

Sample fuzzer output:

```bash
*** perf_fuzzer 0.32-rc0 ***
Linux version 4.15.0-rc9+ x86_64
Processor: Intel 6/94/3
Stopping after 50000
Watchdog enabled with timeout 60s
Will auto-exit if signal storm detected
Seeding RNG from time 1518114761

to reproduce, try:
echo 1 > /proc/sys/kernel/nmi_watchdog
echo 0 > /proc/sys/kernel/perf_event_paranoid
echo 750 > /proc/sys/kernel/perf_event_max_sample_rate

Pid=32534, sleeping 1s

### APPENDIX

#### A. PERF EVENT INTERFACE
The complex interface for the perf_event_open() system call is included below.

The prototype for the system call looks like this:

```c
int perf_event_open(struct perf_event_attr *attr,  
                    pid_t pid, int cpu, int group_fd,  
                    unsigned long flags);
```
C. FUZZER STATUS, LATE 2016

On many modern Linux distributions the perf event interface is disabled by default (it can be re-enabled via the /proc/sys/kernel/perf_event_paranoid file). This is due in part to the numerous bugs we have found, as well as the possibility of side-channel attacks through the detailed timing information provided by the interface.

There are various paranoid settings: -1 means unrestricted, 0 means allow per-cpu system wide data, 1 means allow both kernel and user measurements, 2 means only allow restricted userspace measurements, and there is an out-of-kernel patch applied in many distributions to add a level of “3” to totally disable events [8].

There are a few common warnings that show up at all levels. The first is the warning when the NMI interrupt took too much time leading to throttling: perf: interrupt took too long (3152 > 3135), lowering kernel.perf_event_max_sample_rate to 63250

Another is a WARNING in the breakpoint code which is reproducible and might be an actual bug but no one has bothered to chase it down. WARNING: CPU: 0 PID: 24577 at arch/x86/kernel/hw_breakpoint.c:121 Can’t find any breakpoint slot

Paranoid level 2 should be the safest level, and you would not expect to be able to crash at this level. However as of the 4.9 timeframe it was possible on many architectures, as shown in the summary in Table 6.

Paranoid level 1 also enables kernel events. Crashing tends to happen more quickly here, as seen in Table 7.

Paranoid level 0 enables per-CPU events, including things like uncore, offcore, and RAPL. It exercises a lot more of the unusual CPU event sources. Crashing tends to happen even more quickly here, as seen in Table 8.

We usually avoid fuzzing at level -1, as this enables ftrace/-tracepoints and this triggers “don’t-do-that-then” type bugs, such as inserting an interrupt tracepoint inside of the interrupt handler which then overflows causing recursive interrupts. You can get some pretty spectacular stack traces, and crashes happen quickly as seen in Table 9.

Fuzzing as root has all the fun of -1, but also the perf_fuzzer does some wacky things to various /proc files which can be pretty verbose. This is generally not recommended, but the Linux developers have fixed many of the issues so in general the things preventing this (such as setting impossibly fast interrupt rates) are no longer a problem.

### Table 6: Fuzzer results at paranoid level 2 (user only).

<table>
<thead>
<tr>
<th>Machine</th>
<th>Warnings</th>
<th>time to crash</th>
<th>kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Pentium 4</td>
<td>1</td>
<td>7m49s</td>
<td>4.9-rc0</td>
</tr>
<tr>
<td>Intel Core2</td>
<td>1</td>
<td>n/a (7days+)</td>
<td>4.9-rc0</td>
</tr>
<tr>
<td>Intel Haswell</td>
<td>1</td>
<td>3d9h26m</td>
<td>4.9-rc0</td>
</tr>
<tr>
<td>Intel Skylake</td>
<td>1</td>
<td>7d8h37m</td>
<td>4.9-rc0</td>
</tr>
<tr>
<td>AMD A10</td>
<td>1</td>
<td>2d</td>
<td>4.9-rc0</td>
</tr>
<tr>
<td>Sparc</td>
<td>0</td>
<td>30s</td>
<td>3.2</td>
</tr>
</tbody>
</table>

### Table 7: Fuzzer results at paranoid level 1 (user+kernel).

<table>
<thead>
<tr>
<th>Machine</th>
<th>Warnings</th>
<th>Time to crash</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core2</td>
<td>2</td>
<td>1d15h20m</td>
<td>4.9-rc0</td>
</tr>
<tr>
<td>Intel Haswell</td>
<td>0</td>
<td>21h25m</td>
<td>4.9-rc0</td>
</tr>
<tr>
<td>Intel Skylake</td>
<td>0</td>
<td>n/a (5d+)</td>
<td>4.9-rc0</td>
</tr>
<tr>
<td>AMD A10</td>
<td>0</td>
<td>2h15m</td>
<td>4.9-rc0</td>
</tr>
</tbody>
</table>

### Table 8: Fuzzer results at paranoid level 0 (system-wide/uncore).

<table>
<thead>
<tr>
<th>Machine</th>
<th>Warnings</th>
<th>Time to Crash</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core2</td>
<td>3</td>
<td>21h19m</td>
<td>4.9-rc0</td>
</tr>
<tr>
<td>Intel Haswell</td>
<td>3</td>
<td>8h58m</td>
<td>4.9-rc0</td>
</tr>
<tr>
<td>Intel Skylake</td>
<td>0</td>
<td>4h50m</td>
<td>4.9-rc0</td>
</tr>
<tr>
<td>AMD A10</td>
<td>1</td>
<td>7h55m</td>
<td>4.9-rc0</td>
</tr>
</tbody>
</table>

### Table 9: Fuzzer results at paranoid level -1 (tracepoints too).

<table>
<thead>
<tr>
<th>Machine</th>
<th>Warnings</th>
<th>Time to Crash</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core2</td>
<td>0</td>
<td>14m</td>
<td>4.9-rc0</td>
</tr>
<tr>
<td>Intel Skylake</td>
<td>0</td>
<td>34m</td>
<td>4.9-rc0</td>
</tr>
</tbody>
</table>

such as inserting an interrupt tracepoint inside of the interrupt handler which then overflows causing recursive interrupts. You can get some pretty spectacular stack traces, and crashes happen quickly as seen in Table 9.

Fuzzing as root has all the fun of -1, but also the perf_fuzzer does some wacky things to various /proc files which can be pretty verbose. This is generally not recommended, but the Linux developers have fixed many of the issues so in general the things preventing this (such as setting impossibly fast interrupt rates) are no longer a problem.