SABRINE: State-Based Robustness Testing of Operating Systems

Domenico Cotroneo, Domenico Di Leo, Francesco Fucci, Roberto Natella

Federico II University of Naples
Dipartimento di Ingegneria Elettrica e Tecnologie dell’Informazione (DIETI)

28th IEEE/ACM Intl. Conf. on Automated Software Engineering
Silicon Valley, CA, USA, Nov. 11th-15th, 2013
Mission-critical systems require that OSs should **behave robustly when unexpected and erroneous conditions arise from the environment**

- Invalid or untimely inputs from users and applications
- I/O errors
- Resource exhaustion
- ...

Robust OSs have to **gracefully handle** anomalous events and avoid catastrophic consequences (e.g., OS crashes)

Robustness testing deliberately “injects” anomalous events (“faults”) during the execution of the system
Robustness testing – model-based

- Model-based approaches use a behavioral model of the system (e.g., FSM, IOLTS, ...)
  - for each state of the model, (in)valid inputs and events are defined and injected
  - robustness is checked with respect to the model or to invariants

- These approaches have traditionally been adopted for small systems and protocols

- Unfortunately, they do not scale well for OSs and its components, since they are too large and complex and their internals are often unknown
In **black-box** approaches, robustness test cases are based only on the **input domain**
- e.g., using **out-of-range inputs** and “fuzzing”

To account for the **internal state of the system**, several injections are performed at **random times**
- Many tests are **redundant** as they are performed on the same state
- Many important states may be **missed** by the tests

Using **manually-defined test scenarios** is time-consuming and may still be inaccurate
Contribution: an automated, efficient approach for state-based robustness testing of OSs (SABRINE)
- avoids redundant experiments and improves state coverage
- avoids the manual definition of behavioral models or test scenarios

We implemented and evaluated this approach for robustness testing of a Linux-based RTOS for the avionic domain
Basic idea:

- The internal state of an OS component is determined by the history of interactions occurring at its interfaces.

- From the history of interactions, we extract behavioral models of the OS component.

- Based on the behavioral model, we perform distinct fault injections at each state, to efficiently cover different states of the target.
The Linux kernel includes a fault injector that forces erroneous return codes to simulate failed memory allocations, failed I/O operations, ...

```c
void * kmem_cache_alloc (struct kmem_cache * cachep, gfp_t flags)
{
    void * objp;
    if (should_failslab(cachep, flags))
        return NULL;
    ...
    return objp;
}
```

Faults are injected at random or at a user-defined time.
A component is an OS subsystem that provides services (e.g., memory, process and I/O managers).

Component under test (e.g., a new device driver or filesystem)

Service failure (fault)
Overview of the approach

1. Behavioral Data Collection
2. Pattern Identification and Clustering
3. Behavioral Modeling

Component interactions

Cluster

Sequence

Finite State Automata
First, the OS is executed without injecting faults, using a workload that reflects its expected usage.

We profile the target component by recording:
- service invocations towards and by the target component (input and output interactions)
- service invocations that can fail (injectable interactions)

Component interactions can be probed using kernel debugging tools (e.g., DTrace, SystemTap, ...)

![Diagram of file system components]
An execution log is divided into **sequences**

- i.e., a set of interactions that occur during the same system call, interrupt request, or kernel task execution
- **note**: two executions of the same type are two distinct sequences, and can produce different interactions

Unique sequences (**patterns**) are then extracted
Pattern clustering

- Patterns are often not identical, but exhibit few interactions that are different or that appear in different order
  - This would lead to generate redundant robustness test cases
- Therefore, we derive clusters of similar patterns
  - Each cluster represents a “mode of operation” of the component

**Pattern A**
- ext3_dirty_inode
- journal_start
- kmem_cache_alloc
- __getblk
- journal_get_write_access
- __alloc_pages
- <GAP>
- kmem_cache_alloc
- <GAP>
- journal_stop

**Pattern B**
- ext3_dirty_inode
- journal_start
- kmem_cache_alloc
- __getblk
- journal_get_write_access
- __alloc_pages
- kmem_cache_alloc
- journal_dirty_metadata
- kmem_cache_alloc
- __brelse
- journal_stop

additional memory allocation for writing file metadata
1. A similarity score is computed between each pair of patterns (Smith-Waterman algorithm)
   - It first searches the best alignment between two patterns
   - The score is higher when there are many matching symbols and few gaps/mismatches

2. Similar patterns are grouped (spectral clustering)
   - Patterns are the nodes of a weighted graph, and the similarity score is the weight of the edge between two nodes
   - By cutting “weak” edges, the graph is split into partitions that are “strongly connected” (i.e., very similar patterns)
1. For each cluster, we obtain a behavioral model (kBehavior algorithm)
   - A Finite State Automaton (FSA) is incrementally extended with new transitions and states
   - Transitions represent interactions of the patterns

2. A robustness test case is generated for each injectable interaction included in the FSA
   - This allows to perform injections in different contexts
An FSA is derived from a cluster containing two patterns

A robustness test case is generated for each injectable interaction in the model
To perform a robustness test case, the system is again executed with the same workload

- We automatically generate a kernel injection module that keeps track of the OS state at run-time
- When the injector notices that an injectable function is invoked at a given state, it forces an erroneous return code from that function call
Experimental evaluation (1)

- SABRINE was evaluated on a **Linux-based OS** (FIN.X-RTOS), compliant to the **DO-178B** safety guidelines.
- We applied SABRINE to enhance the fault injection framework of the Linux kernel.
  - Faults are injected at the **memory allocator** interface.
  - We targeted **EXT3**, **ReiserFS**, and the **SCSI** subsystem.
SABRINE was compared with the "random" approach adopted by the default Linux fault injector

- Invocations of the injectable function (\texttt{kmem\_cache\_alloc}) fail with a fixed probability $P=10\%$
- 1,000 random injections are performed for each component

The OS fails when it is crashed or its state is corrupted

- To detect state corruptions in the OS, we enabled several consistency checks in the kernel (stack overflows, stuck system calls, locks not released, ...)

## Test generation

<table>
<thead>
<tr>
<th>Clusters (EXT3) Behavior</th>
<th>Context</th>
<th># patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 gets and sets the file metadata</td>
<td>stat syscall</td>
<td>6</td>
</tr>
<tr>
<td>2 retrieves and stores in memory the file index block, or updates it on the disk</td>
<td>open, unlink syscalls</td>
<td>5</td>
</tr>
<tr>
<td>3 copies file contents from disk to a cache, and modifies it</td>
<td>write syscall</td>
<td>8</td>
</tr>
<tr>
<td>4 modifies the contents of a file already in the disk cache</td>
<td>write syscall</td>
<td>8</td>
</tr>
<tr>
<td>5 copies a large amount of data from a file to a socket</td>
<td>sendfile syscall</td>
<td>12</td>
</tr>
<tr>
<td>6 copies a small amount of data from a file to a socket</td>
<td>sendfile syscall</td>
<td>10</td>
</tr>
<tr>
<td>7 flushes a small amount of data from the cache to the disk</td>
<td>pdflush kernel task</td>
<td>19</td>
</tr>
<tr>
<td>8 flushes a large amount of data from the cache to the disk</td>
<td>pdflush kernel task</td>
<td>6</td>
</tr>
<tr>
<td>9 updates file metadata to reflect that is has been memory-mapped</td>
<td>mmap2 syscall</td>
<td>5</td>
</tr>
</tbody>
</table>
Robustness vulnerabilities

- Both types of injections provoked several OS and application failures.
- All OS failures were always caused by two robustness "vulnerabilities", that affected both EXT3 and ReiserFS (radix_tree_node_alloc and __get_blk).

### Stack frame

<table>
<thead>
<tr>
<th>Stack frame</th>
<th>Kernel function</th>
<th>Injection</th>
<th>Call by EXT3</th>
<th>System call</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\texttt{kmem_cache_alloc}+0x22/0x110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>radix_tree_node_alloc+0x35/0xb0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>radix_tree_insert+0x16e/0x1d0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add_to_page_cache+0x65/0x1d0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>add_to_page_cache_lru+0x1b/0x40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>mpage_readpages+0x70/0xe0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>\texttt{ext3_readpages}+0x19/0x20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>\texttt{__do_page_cache_readahead}+0x176/0x210</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ondemand_readahead+0xbe/0x170</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>page_cache_async_readahead+0x66/0x90</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>generic_file_splice_read+0x4a9/0x630</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>do_splice_to+0x61/0x80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>splice_direct_to_actor+0x8f/0x180</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>do_splice_direct+0x3b/0x60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>do_sendfile+0x187/0x240</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>\texttt{sys_sendfile64}+0x77/0xa0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>sysenter_past_esp+0x5f/0x91</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Efficiency and reproducibility

- Only a minority of random injections hit the two robustness vulnerabilities
- Using SABRINE, the same vulnerabilities can be found with much less tests (77 vs 2,000)
- SABRINE tests exhibit an high average probability of reproducing OS failures

**Vulnerable functions**

<table>
<thead>
<tr>
<th></th>
<th>Random</th>
<th>SABRINE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>__get_blk</strong></td>
<td>29.0%</td>
<td>68.8%</td>
</tr>
<tr>
<td><strong>radix_tree_node_alloc</strong></td>
<td>3.8%</td>
<td>77.7%</td>
</tr>
</tbody>
</table>

**EXT3**

- 68.8% 77.7%

**ReiserFS**

- 100.0% 100.0%

- 0.2% 9.4%
Conclusion

- The state of the system at the time of the injection plays an important role in robustness testing.
- Randomly-timed injections are inefficient at covering “rare” states.
- SABRINE makes robustness testing more systematic and efficient with little effort.
Thank you

My contacts:
roberto.natella@unina.it