ALICE: Application-Level Intelligent Crash Explorer

This is a user documentation of the ALICE tool, which can be used to discover "crash vulnerabilities" in applications. Crash vulnerabilities are problems that get exposed by a sudden power loss or system crash while the application is running, and the application cannot recover correctly after rebooting the machine. ALICE focuses on single-node applications that run atop file systems. ALICE is different from similar tools in that it aims to find vulnerabilities that might occur across all file systems, including future ones. ALICE is also unique in targeting vulnerabilities associated with different source-code lines of the application, instead of checking the application's correctness atop arbitrarily (or systematically, but less useful) simulated crash scenarios. ALICE is designed to be extensible: both how it checks different source lines, and the combined behavior it assumes of underlying file systems, can be customized. The ALICE tool is a by-product of a research project (`http://research.cs.wisc.edu/adsl/Publications/alice-osdi14.html`) in the University of Wisconsin-Madison.

Chapter 1

Installation

ALICE was tested to work on Ubuntu-12.02, and should be expected to work on similar (i.e., Linux-like) operating systems. The following are specific requirements:

1. Python-2.7, as the default version of python invoked via `/usr/bin/env python`.
2. Standard software build tools, such as `gcc` and `GNU Make`.
3. The libunwind libraries, installable in Ubuntu-12.02 using `apt-get install libunwind7`.

The following are the steps to install ALICE:

1. Download the most recent source-code tarball of ALICE, and untar it. This should produce a directory named `alice`.
2. Set the environmental variable `ALICE_HOME` to point to the `alice` directory (i.e., the untared directory). For example, this can be done by adding the line `export ALICE_HOME=/wherever-untarred/alice` to your `.bashrc` file.
3. Set the `PATH` environmental variable to include the `alice/bin` directory. For example, this can be done by adding the line `export PATH=$PATH: /wherever-untarred/alice/bin` to your `.bashrc` file.
4. Install the `alice-strace` tracing framework by moving into the `alice/alice-strace` directory, and running `./configure; make; make install;`

Chapter 2

Basic Usage

The typical workflow for using ALICE has two steps. First, an application workload is run, and a trace of its activities are recorded. Second, ALICE is given this trace and a *checker* script (explained later); ALICE explores the trace and displays discovered vulnerabilities. This documentation explains the typical usage of ALICE by using a toy application.

Toy application

The toy application can be found in `alice/example/toy/toy.c`; the reader is encouraged to go through it. The application does the following:

1. It updates a file called `file1`, changing the contents of the file from *"hello"* to *"world"*. The update is done using a typical "write to temporary file and rename" sequence, so that the contents are updated atomically. Immediately after updating, the application prints a message to the user's terminal (the user can then supposedly assume that the file has been updated, and that the file will contain *"world"* even if a power loss happens).

2. It creates two links to the file, `link1` and `link2`. The (imaginary) semantics of the toy application require both these links to be created atomically (i.e., if a power loss happens, either both links exist or neither do not).

Step 1: Running the application and recording a trace

A script that runs the application and records a trace, along with all initialization setup, can be found in `alice/example/toy/toy_workload.sh`; the reader is encouraged to go through it.

To perform Step 1, two directories are needed. The first, the *workload directory*, is where the files of the application will be stored. The application, as it runs, will modify the workload directory and its contents. For the toy application, this is the place where `file1`, `link1`, and `link2`, are placed. The

`toy_workload.sh` script first creates the workload directory, `workload_dir`, and then initializes it with the file `file1` containing *"hello"*.

The other needed directory, *traces directory* is for storing the (multiple) traces that are recorded as the application is run. The `toy_workload.sh` script next creates this directory, `traces_dir`. After setting up the workload directory and the traces directory, the `toy_workload.sh` script does a few more initialization things: compiling the `toy.c` application, and `cd`ing into `workload_dir` so that the toy application can be run within there.

The `toy_workload.sh` script finally runs the application and records traces, by issuing the following command:

```
alice-record --workload_dir . \
         --traces_dir ../traces_dir \
         ../a.out
```

If the reader is familiar with the *strace* utility, the above command is similar to an invocation of strace: `alice-record` is a script that records traces, while `../a.out` is the actual application to be run (the process and all subprocesses of `../a.out` are traced, similar to strace with the `-ff` option). The `alice-record` script requires two mandatory arguments: the workload directory and the traces directory (`alice-record` takes one more optional argument, `--verbose`, to control verbosity).

Step 2: Supply ALICE with the trace and the checker, and get back list of vulnerabilities

Step 2 requires the user to supply ALICE with a checker script. The checker script will be invoked multiple times by ALICE, each invocation corresponding to a (simulated) system crash scenario that could have happened while the application was running in Step 1. During each invocation, the checker script will be given a directory that reflects the state of the workload directory if the (simulated) crash had really happened. If the given crashed-state workload directory has an expected (i.e., consistent) set of files, the checker script should exit with status zero, and should exit with a non-zero status otherwise.

ALICE supplies the checker script with two command-line arguments. The first is the path to the crashed-state workload directory. The second command-line argument to the checker script is the path to an *stdout file*. The stdout file contains all the messages that had been printed to the user's terminal at the time of the crash (corresponding to the supplied crashed-state workload directory), and can be used by the checker to check for durability, as explained below. Note that the crashed-state workload directory supplied by ALICE might differ from the original workload directory in Step 1. Hence, for applications that expect the *absolute path* of the contents within the workload directory to not have changed (a small subset of applications in our experience), the checker script needs to move the supplied directory to the original directory, and then operate atop the original directory.

The checker script for the toy application can be found in `alice/example/toy/toy_checker.py`, and the reader is encouraged to go through it. The script first changes the current working directory into the crashed-state directory supplied by ALICE, and reads all the messages printed in the terminal at the time of the crash by reading the stdout file supplied by ALICE. If the application has printed the *"Updated file1 to* world*"* message, the checker script makes sure that `file1` contains *"world"*; otherwise, the checker script makes sure that `file1` contains either *"hello"* or *"world"*. The checker script then makes sure that either `link1` and `link2` are both present, or are both not present. If any of the checked conditions do not hold, the checker script results in an assertion failure, thus exiting with a non-zero status (and thus informing ALICE that the application will fail if the simulated crash scenario happens in real).

After writing the checker script, the user can invoke the `alice-check` script to actually run ALICE and get the list of vulnerabilities. The reader is encouraged to run the following command from within the `alice/example/toy` directory, to get a list of vulnerabilities discovered in the toy application (after running `toy_workload.sh` first).

```
alice-check --traces_dir=traces_dir --checker=./toy_checker.py
```

The `alice-check` script has the following arguments:

traces_dir Mandatory. The traces directory, from Step 1.

checker Mandatory. The checker script.

threads Optional, default is 4. ALICE invokes checker scripts parallely, each checker script given a separate crashed-state directory to work on. Some applications do not allow multiple simultaneous invocations, and might require this option to be set to 1.

debug_level Optional, default is 0. Verbosity of warnings, can be 0, 1, or 2.

ignore_mmap Optional, default is False. The current version of ALICE does not trace `mmap`-writes, and cannot correctly work with application workloads that use memory mapping to modify relevant files (see caveats and limitations). If the recorded trace during Step 1 involves a writeable `mmap()` to a seemingly relevant file, `alice-check` aborts execution by default. However, some application workloads use `mmap()` only on files that are irrelevant to crash consistency, for example to implement a shared-memory lock dealing with multi-process concurrency synchronization. This option can be set to True if the user is sure that the `mmap()`s observed while running the application workload are irrelevant to finding crash vulnerabilities. Some database applications use `mmap()` for concurrency control, even when configured not to use `mmap()` for otherwise accessing files, and require this option.

Understanding ALICE's output

ALICE first outputs a list of list of the logical operations that form the *update protocol* used by the application workload invoked in Step 1. The logical operations displayed is similar to a system-call trace, except that it is easier to understand, for example substituiting file names instead of file descriptor numbers.

ALICE then displays any discovered vulnerabilities. Vulnerabilities are displayed in two ways: *dynamic vulnerabilities*, relating to different operations in the update protocol, and *static vulnerabilities*, relating to source-code lines. The proper display of static vulnerabilities requires the originally traced application to have debugging symbols; also, ALICE associates each logical operation to one of the stack frames in the logical operation's stack trace to display static vulnerabilities, and this association can sometimes be faulty.

Chapter 3

Customizing, Extending, and Hacking

ALICE is designed to be extensible. The current version of ALICE strips off many features that were previously implemented, in hopes that a smaller code base promotes extensions. However, the current version is also not sufficiently commented, and does not follow some good coding practices; a well-commented version of the software might be released in the future if users shows interest.

To extend ALICE, readers are required to go through our publication (`http://research.cs.wisc.edu/adsl/Publications/alice-osdi14.html`) to understand ALICE's design and philosophy. Note that there is some terminology difference between the publication and ALICE's source code; in particular, *logical operations* discussed in the publication correspond to *micro operations* in the source code, while *micro operations* in the publication correspond to *disk operations* in the source code.

ALICE's default exploration strategy, which investigates the ordering and atomicity of each system call and reports any associated vulnerabilities, is coded in `alice/alicedefaultexplorer.py`, and can be easily changed. The `alicedefaultexplorer.py` code is complicated since it displays static vulnerabilities and invokes checkers in multiple threads. A functionally equivalent exploration strategy can be simpler.

ALICE's default APM is coded in `alice/alicedefaultfs.py`, and can be easily changed. The `alicedefaultfs.py` code is complicated since it models a file system that can be configured to split file operations in different granularities. A functionally equivalent file system (with a single granularity) can be simpler.

Other than the extensions discussed till now, users might try to add support for more system calls, file attributes, symbolic links, or other such details, in ALICE. Relevant to these, the `_aliceparsesyscalls.py` script contains code that converts system calls into logical operations, while the `replay_disk_ops()` function from the `alice.py` script contains code that re-constructs a directory from a given list of micro-ops.

Chapter 4

Caveats and Limitations

ALICE is a *safe*, but not a *complete* tool. That is, the application might have additional vulnerabilities than those discovered and reported. ALICE is thus not aligned towards comparing the correctness of different applications; specifically, any comparisons when not using equivalent workloads and checkers can easily produce confusing, wrong inferences. Also, any vulnerability displayed by ALICE might already be known to an application developer: the application documentation might explicitly require that the underlying file system not behave in those ways that will expose the vulnerability, or might simply not provide those guarantees that are being checked by the checker.

The default file-system model (APM) used by ALICE is designed to also find vulnerabilities that can get exposed by future file systems; some crash scenarios that are possible with the default model do not happen in common current file systems. Also, ALICE's output (a list of vulnerabilities) is only designed to show the number of source lines that require ordering or atomicity. It is thus erraneous to directly correlate the number of vulnerabilities shown by ALICE with current real-world impact.

ALICE does not currently attempt to deal with any file attributes (including modification time) other than the file size, or with the `FD_CLOEXEC` and `O_CLOEXEC` facilities. If the application's logic (that is invoked in the workload and the checker) depends on these, ALICE's output is probably wrong. Support for a few rarely-used system calls is also lacking; warning or error messages are displayed by ALICE if the application workload had invoked such calls. The situation for symlinks is similar; while the current version of ALICE attempts to support them slightly, if the application logic depends on symlinks, ALICE's output might be wrong.

The current version of ALICE also does not support tracing memory-mapped writes; applications that use such writes as a part of their (relevant) update protocol cannot use ALICE. Note that a version of ALICE used in our published research paper (`http://research.cs.wisc.edu/adsl/Publications/alice-osdi14.html`) traced memory-mapped writes, but support was removed in the interest of distributability.

Adding support for file attributes, `CLOEXEC`, symlinks, and `mmap()` writes does not require any changes to the design of ALICE, and might be done in future versions if users deem them helpful.

Chapter 5

Credits, Acknowledgements, and Contact Information

Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, and Samer Al-Kiswany were involved in various aspects of design, coding, and testing of the ALICE tool. Thanumalayan Sankaranarayana Pillai (madthanu@cs.wisc.edu) is the primary author, and might serve to be the best contact for bug reports, feature requests, or other general discussions.

The ALICE tool is a by-product of a research project (`http://research.cs.wisc.edu/adsl/Publications/alice-osdi14.html`) in the University of Wisconsin-Madison, and due credit must be given to all parties who were involved in or contributed to the project.

The `alice-strace` tracing framework is a slight customization of the strace tool (`http://sourceforge.net/projects/strace/`), along with some code adapted from strace-plus (`https://code.google.com/p/strace-plus/`). Credits must be given to the authors and contributors of strace and strace-plus.