# ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level

Jan Fiedor, Tomáš Vojnar

Brno University of Technology (BUT)

RV, September 26, 2012

# Introduction

- Why to analyse multi-threaded C/C++ programs?
  - There is an ever-growing number of such programs
  - Concurrency errors easy to create, but difficult to discover
  - Many tools for Java (IBM ConTest, RoadRunner, CalFuzzer, Chord, etc.), however, not so many for C/C++ (Fjalar)
- Dynamic analysis
  - Extrapolates the witnessed behaviour
  - May detect errors not witnessed in the given execution
  - Needs to monitor the execution of a program (difficult)
- ANaConDA framework
  - A framework simplifying the creation of dynamic analysers
    - Monitors the execution of a multi-threaded C/C++ program
    - Offers notification about important events to the analyser
  - Supports noise injection

# Information Provided by the Framework

Memory access information:

- Reads, writes and atomic updates

Synchronisation information:

- Lock acquisitions/releases, signaling conditions and waiting on them

Thread information:

- Thread started/finished

Exception information:

- Exception thrown/caught

Convenient information for localising detected errors:

- Backtraces (containing return addresses)

# A Simple Analyser Monitoring Lock Operations

```
PLUGIN_INIT_FUNCTION()
{
  // Register a callback function called before a lock is released
  SYNC_BeforeLockRelease(beforeLockRelease);

  // Register a callback function called after a lock is acquired
  SYNC_AfterLockAcquire(afterLockAcquire);
}

VOID beforeLockRelease(THREADID tid, LOCK lock)
{
  CONSOLE("Before lock released: thread " + decstr(tid) + ", lock " + lock + "\n");
}

VOID afterLockAcquire(THREADID tid, LOCK lock)
{
  CONSOLE("After lock acquired: thread " + decstr(tid) + ", lock " + lock + "\n");
}
```

# Instantiation for a Specific Multithreading Library

The framework might be configured to work with various multithreading libraries:

- The same dynamic analyser may be reused for different libraries
- Currently instantiated for the pthread library and Win32 API

The user must specify:

- Which functions perform certain types of thread-related operations
  - Specify the names of these functions
- Which arguments represent the synchronisation resources
  - Specify the indices of these arguments
- How to transform synchronisation resources to their abstract identifications
  - Specify the mapper objects used to perform the transformation

# Instantiation for the Pthread Library

To configure monitoring of lock acquisitions:

- The required information must be added to `conf/hooks/lock`

```
# <function name>       <index>  <mapper object>
pthread_mutex_lock      1        addr()
pthread_mutex_trylock   1        addr()
```

To configure monitoring of lock releases:

- The required information must be added to `conf/hooks/unlock`

```
# <function name>             <index>  <mapper object>
__pthread_mutex_unlock_usercnt 1        addr()
```

# Noise Injection Support

Noise Injection Techniques:

- Aim at increasing the number of different witnessed interleavings
- Disturb the scheduling of threads by inserting noise generating code
  - e.g., by inserting calls of `yield` or `sleep`
- Force the program to switch threads at times it would normally seldom do it

User may typically influence:

- Type of noise (currently *sleep* and *yield* noise is supported)
- Noise frequency (how often the noise should occur)
- Noise strength (how strong the noise should be)

# Configuring Noise Injection

The framework supports fine-grained combinations of noise, i.e., different noise injection settings might be used for:

- Reads, writes and atomic updates (configured in `anaconda.conf`)
- Each of the monitored functions (configured in `lock`, `unlock` etc.)

An example how to use different noise settings for reads and writes:

```
[noise]          # Global noise settings
type = yield     # Insert calls to yield
frequency = 100  # Inject noise in 10 % of times
strength = 4     # Give up the CPU 4 times
[noise.read]     # Noise settings for read accesses
type = yield     # Insert calls to yield
frequency = 200  # Inject noise in 20 % of times
strength = 8     # Give up the CPU 8 times
[noise.write]    # Noise settings for write accesses
type = sleep     # Insert calls to sleep
frequency = 400  # Inject noise in 40 % of times
strength = 2     # Sleep for 2 milliseconds
```

# Experiments

ANaConDA is a *pintool* (plugin for the PIN framework):

- Instrumentation done in the memory (transparent use of libraries)
- Can handle generated and self-modifying code
- May be used on both Linux and Windows
- Slowdown of the execution is around 100 times

Firefox 10 browser

- So far without a test harness
- Found several known data races considered as harmless
- Proved that the tool can handle even very large programs

Unicap libraries: libraries for concurrent video processing

- Found several previously unknown data races
- Some of them cause programs using these libraries to crash

# Conclusion

- We have presented ANaConDA
  - A framework simplifying the creation of dynamic analysers
- We have shown
  - How to write an analyser
  - How to instantiate the framework for a particular multithreading library
  - How to configure and use the noise injection

# Future Work

- Improvements to the framework
  - Additional notifications (fork/join notifications etc.)
  - Better access to debugging information
  - Smarter instrumentation
  - ...
- More experiments
- More sophisticated types of noises
- New detectors for concurrency errors

# End of presentation

Thank you for your attention!

ANaConDA framework available at:

**http://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda/**

# An Example How to Obtain a Backtrace

```
VOID beforeMemoryWrite(THREADID tid, ADDRINT addr, UINT32 size,
  const VARIABLE& variable, const LOCATION& location)
{
  // Helper variables
  Backtrace bt;
  Symbols symbols;

  // Get the backtrace of the current thread
  THREAD_GetBacktrace(tid, bt);
  // Translate the return addresses to locations
  THREAD_GetBacktraceSymbols(bt, symbols);

  CONSOLE_NOPREFIX("Thread " + decstr(tid) + " backtrace:\n");

  for (Symbols::size_type i = 0; i < symbols.size(); i++)
  { // Print information about each return address in the backtrace
    CONSOLE_NOPREFIX("    #" + decstr(i) + (i > 10 ? " " : "  ")
      + symbols[i] + "\n");
  }
}
```

# Choosing Parts of a Program To Be Monitored

Any image (executable file, library) can be:

- Excluded from instrumentation (`conf/filters/ins/exclude`)
- Forced to be instrumented (`conf/filters/ins/include`)

An example how to configure the framework to not instrument libraries:

```
# Do not instrument standard Linux libraries
/lib/*
/lib64/*
/usr/lib/*
/usr/local/lib/*
# Do not instrument standard Windows libraries
${windir}/system32/*
${windir}/SYSTEM32/*
# Do not instrument PIN libraries
${PIN_HOME}/*
# Do not instrument ANaConDA plugins
*/lib/ia32/*
*/lib/intel64/*
```

# Performing an Analysis

To analyse a multi-threaded C/C++ program using the framework, one can use the `anaconda.sh` (`anaconda.bat`) script:

```
./anaconda.[sh|bat] <path-to-analyser> <path-to-program>
```

To pass additional parameters to PIN or ANaConDA, one has to use the following command:

```
$PIN_HOME/pin[.bat] <pin-args> -t <path-to-anaconda>
 -a <path-to-analyser> <anaconda-args>
 -- <path-to-program> <program-args>
```