Noise-Based Testing and Analysis of Multi-threaded C/C++ Programs on the Binary Level

Jan Fiedor, Tomáš Vojnar

Brno University of Technology (BUT)

PADTAD, July 16, 2012
Plan of the talk

1. Introduction
2. Monitoring Multi-threaded C/C++ Programs
3. Fine-Grained Combinations of Noise
4. Experiments
5. Conclusion
Introduction

- **Testing**
  - One of the most common ways to discover errors
  - Detects only errors witnessed in the given execution
  - Many repetitions needed due to the non-deterministic thread scheduling
  - When done naively, the repeated execution needs not differ much, and many errors may be missed

- **Dynamic analysis**
  - Extrapolates the witnessed behaviour
  - May detect errors not witnessed in the given execution
  - Needs to insert some monitoring code into the program

- **Noise injection**
  - Disturbs the scheduling of threads to see uncommon executions
  - Increases the chances to detect errors
  - Useful for both testing and dynamic analysis
Plan of the talk

1. Introduction

2. Monitoring Multi-threaded C/C++ Programs

3. Fine-Grained Combinations of Noise

4. Experiments

5. Conclusion
Monitoring (and noise injection) code might be inserted on several levels:

- **Source code level**
  - Code inserted to the source code before compilation
- **Level of the intermediate code**
  - Code inserted to the compiler’s intermediate code during compilation
- **Binary level**
  - Code inserted to the program’s binary after compilation

On the binary level, the monitoring code might be inserted:

- By modifying the binary of a program before it is executed
  - Static binary instrumentation
- By modifying the binary at the run-time in the memory
  - Dynamic binary instrumentation
We use dynamic binary instrumentation to insert the monitoring code:

- **Advantages:**
  - No need to have the source code of the program
  - More precise (insertion after all optimisations)
  - More transparent (no need to have 2 separate versions of libraries)
  - Easy handling of assembly code inserted to C/C++ code
  - Easy access to low level information (e.g. register allocations)
  - Can handle self-generating and self-modifying code

- **Disadvantages:**
  - Slower (than using static instrumentation)
    - Code must be inserted before every execution
    - Code is usually executed in some kind of low-level virtual machine
  - Problematic access to higher level information (e.g. names of variables)
Monitoring Multi-threaded C/C++ Programs

What we need to monitor:

- **Threads** (creation, termination)
  - Usually done by calling suitable library functions
- **Synchronisation** among threads (lock, unlock, wait, signal)
  - Usually done by calling suitable library functions
- **Memory accesses** (reads and writes)
  - Performed by instructions

The analyser should be **notified**:

- When some event is about to happen (**before** notifications)
- When some event just happened (**after** notifications)
Monitoring Execution of Functions

Naïve approach: instrument appropriate call instructions
- Must analyse all call instructions in the binary and its libraries
- Code inserted after call instructions might not be executed

Better approach: wrap the monitored functions in other functions
- Must know the signature of the original function
- Calling the original function from the wrapper function might be slow

Quicker approach: instrument the code of the monitored functions
- Insert the monitoring code
  - Before the first instruction of the function
  - Before every return instruction in the function
- Decreases the instrumentation overhead
- More generic (no need to know signatures etc.)
At the binary level, it is possible to return from a function even from code NOT belonging to that function:

Program’s binary

```
401113:  mov $0×602540,%edi
401118:  callq 400e80 <unlock>
40111d:  test %eax,%eax
```

pthread library

```
ae20 <unlock>:
  ae20:  mov $0x1, %esi
  ae25:  jmpq ad70 <unlock_usr>

ad70 <unlock_usr>:
  ad70:  mov %rdi, %rdx
  ... : ...
  ada5:  xor %eax, %eax
  ada7:  retq
```

We called unlock, but returned from unlock_usr!
Monitoring Execution of Functions: A Solution

Idea:
- Functions usually do not jump outside of the code of the library itself

Solution:
- Insert the monitoring code before every return instruction in the library
- Save the current state of the thread’s call stack before the monitored function is executed (value of the stack pointer is sufficient)
- Before a return instruction is executed
  - Compare the current and previous value of the stack pointer
  - Issue a notification if the values match

Exception:
- Functions from the Win32 API’s kernel32.dll library may jump to the kernelbase.dll library
Monitoring Special Types of Instructions

**Atomic instructions (e.g. xadd):**
- Access memory more than once
- The monitoring code should issue a special notification informing the analyser that some memory accesses happened atomically

**Conditional and repeatable instructions (e.g. rep stos):**
- Most of them access memory
- Might be executed:
  - A fixed number of times
  - Until a condition is met
  - Not at all
- The monitoring code must ensure that the notification is issued as many times as the access actually happened (possibly not at all!)
Abstracting Synchronisation Primitives

Thread management and synchronisation in C/C++:
- Usually done by calling suitable library functions
- Many different libraries can be used for this purpose

To allow dynamic analysers to be reused with multiple libraries:
- A support for abstracting the low-level details is needed

The abstraction can hardly be fully automated—we require the user to specify:
- Which functions perform certain types of thread-related operations
- Which arguments represent the synchronisation resources
- How to transform synchronisation resources to their abstract identifications
Plan of the talk

1. Introduction
2. Monitoring Multi-threaded C/C++ Programs
3. Fine-Grained Combinations of Noise
4. Experiments
5. Conclusion
Noise Injection Basics

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 (e.g., *sleep* or *yield* noise)
- Noise **frequency** (how often the noise should occur)
- Noise **strength** (how strong the noise should be)
Fine-Grained Combinations of Noise

Idea (for data races):
- Data races arise when there are two unsynchronised accesses to the same memory location and at least one of the accesses is a write access.
- When we encounter a memory access, the best we can do is to search the other threads for the second (conflicting) access.

Using the same settings for all accesses:
- The *yield* noise
  - Only a small part of the executions of the other threads is searched.
- The *sleep* noise
  - Blocks the execution of the thread performing the first access.
  - Gives us more time to search the other threads for the second access.
Fine-Grained Combinations of Noise

Idea (for data races) revisited:

- The *sleep* noise seems **better** than the *yield* noise, however:
  - It blocks **not only** the thread performing the first access, but also the threads we want to **search** for the second (conflicting) access
  - Injecting a **larger** amount of the *sleep* noise may considerably **slow down** the execution

- Lower the amount of noise injected into the **other threads** so they perform more memory accesses

- The **two** unsynchronised accesses are often **different types** of memory accesses (one must be **write**, the other is often **read**)

- Lower the amount of noise injected into the **other threads** by using **different** settings for different **types** of memory accesses (reads/writes)
Useful Combinations of Noise (for Data Races)

Use the *sleep* noise only, but with different values of strength:
- Use *bigger* strength for one type of memory accesses
- Use considerably lower strength for the other type of memory accesses
- Still blocks the other threads, just a bit less than before

Use different types of noises:
- Use the *sleep* noise for one type of memory accesses
- Use the *yield* noise for the other type of memory accesses
- Does not block the other threads much
- Forces the program to switch threads more often
- Helps more threads to perform more memory accesses
Plan of the talk

1. Introduction
2. Monitoring Multi-threaded C/C++ Programs
3. Fine-Grained Combinations of Noise
4. Experiments
5. Conclusion
Experiments

We used 116 multi-threaded C/C++ programs for the experiments:

- Student programs implementing a simple ticket algorithm
- Use the pthread library for thread management and synchronisation
- Found errors in around 20% of them (most of them rated full points)

We focused on detection of:

- Data races (wrong synchronisation of accesses to shared variables)
  - Used noise injection in conjunction with dynamic analysis
  - Used a simple AtomRace detector to detect data races
- Assertion errors (erroneous usage of the pthread library)
  - Used noise injection in conjunction with normal testing
Interesting Results for Data Races

Using **too much** noise may actually **suppress** the errors

<table>
<thead>
<tr>
<th>Noise configuration \ Program</th>
<th>t01</th>
<th>t02</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>instrumented, no sleep or yield noise</em></td>
<td>2.4</td>
<td>11.8</td>
</tr>
<tr>
<td>sleep (50% frequency, 10 ms of sleep)</td>
<td>69.2</td>
<td>46.6</td>
</tr>
<tr>
<td>sleep (10% frequency, 10 ms of sleep)</td>
<td>64.0</td>
<td>69.2</td>
</tr>
<tr>
<td>rs-sleep (50% frequency, 0–10 ms of sleep)</td>
<td>96.4</td>
<td>87.8</td>
</tr>
<tr>
<td>rs-sleep (10% frequency, 0–10 ms of sleep)</td>
<td>21.4</td>
<td>55.8</td>
</tr>
<tr>
<td>sleep (50% frequency, 10 ms of sleep) / read 20 ms / write 5 ms</td>
<td>64.8</td>
<td>89.4</td>
</tr>
<tr>
<td>sleep (10% frequency, 10 ms of sleep) / read sleep / write yield</td>
<td>34.2</td>
<td>81.0</td>
</tr>
</tbody>
</table>

To deal with this problem, one may:

- **Lower** the frequency (or strength)
- Use **random** strength instead of a fixed one
- Use **different** noise injection **settings** for different locations
Interesting Results for Data Races

Using different noise injection settings also helps in many other cases

<table>
<thead>
<tr>
<th>Noise configuration \ Program</th>
<th>t06</th>
<th>t07</th>
</tr>
</thead>
<tbody>
<tr>
<td>instrumented, no sleep or yield noise</td>
<td>1.0</td>
<td>1.6</td>
</tr>
<tr>
<td>sleep (50% frequency, 10 ms of sleep)</td>
<td>53.6</td>
<td>69.4</td>
</tr>
<tr>
<td>sleep (10% frequency, 10 ms of sleep)</td>
<td>40.2</td>
<td>70.4</td>
</tr>
<tr>
<td>rs-sleep (50% frequency, 0–10 ms of sleep)</td>
<td>31.0</td>
<td>79.0</td>
</tr>
<tr>
<td>sleep (50% frequency, 10 ms of sleep) / read 5 ms / write 20 ms</td>
<td>92.6</td>
<td>96.2</td>
</tr>
<tr>
<td>yield (50% frequency, 10 calls of yield) / read yield / write sleep</td>
<td>95.0</td>
<td>99.6</td>
</tr>
</tbody>
</table>

Different noise injection settings can be used to speed up the execution

<table>
<thead>
<tr>
<th>Noise configuration \ Program</th>
<th>t04</th>
</tr>
</thead>
<tbody>
<tr>
<td>instrumented, no sleep or yield noise</td>
<td>1.2</td>
</tr>
<tr>
<td>sleep (50% frequency, 10 ms of sleep)</td>
<td>100.0</td>
</tr>
<tr>
<td>sleep (10% frequency, 10 ms of sleep)</td>
<td>56.0</td>
</tr>
<tr>
<td>rs-sleep (50% frequency 0–10 ms of sleep)</td>
<td>86.2</td>
</tr>
<tr>
<td>rs-sleep (10% frequency, 0–10 ms of sleep)</td>
<td>11.8</td>
</tr>
<tr>
<td>sleep (50% frequency, 10 ms of sleep) / read yield / write sleep</td>
<td>100.0</td>
</tr>
<tr>
<td>sleep (10% frequency, 10 ms of sleep) / read yield / write sleep</td>
<td>96.8</td>
</tr>
</tbody>
</table>
Interesting Results for Data Races

Different noise injection settings are sometimes the only thing that helps

<table>
<thead>
<tr>
<th>Noise configuration</th>
<th>Program</th>
<th>t05</th>
<th>t04</th>
</tr>
</thead>
<tbody>
<tr>
<td>instrumented, no sleep or yield noise</td>
<td>t05</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>sleep (50% frequency, 10 ms of sleep)</td>
<td>t05</td>
<td>1.2</td>
<td>7.4</td>
</tr>
<tr>
<td>sleep (10% frequency, 10 ms of sleep)</td>
<td>t05</td>
<td>5.4</td>
<td>96.8</td>
</tr>
<tr>
<td>rs-sleep (50% frequency, 0–10 ms of sleep)</td>
<td>t05</td>
<td>0.6</td>
<td>6.2</td>
</tr>
<tr>
<td>rs-sleep (10% frequency, 0–10 ms of sleep)</td>
<td>t05</td>
<td>0.0</td>
<td>94.4</td>
</tr>
<tr>
<td>sleep (50% frequency, 10 ms of sleep) / read 5 ms / write 20 ms</td>
<td>t05</td>
<td>43.0</td>
<td>62.4</td>
</tr>
<tr>
<td>sleep (10% frequency, 10 ms of sleep) / read sleep / write yield</td>
<td>t05</td>
<td>62.4</td>
<td>9.6</td>
</tr>
</tbody>
</table>

It is better to inject stronger noise before rarer accesses

<table>
<thead>
<tr>
<th>Noise configuration</th>
<th>Program</th>
<th>t04</th>
<th>t05</th>
</tr>
</thead>
<tbody>
<tr>
<td>instrumented, no sleep or yield noise</td>
<td>t04</td>
<td>1.2</td>
<td>64.4</td>
</tr>
<tr>
<td>sleep (10% frequency, 10 ms of sleep) / read sleep / write yield</td>
<td>t04</td>
<td>7.4</td>
<td>64.4</td>
</tr>
<tr>
<td>sleep (10% frequency, 10 ms of sleep) / read yield / write sleep</td>
<td>t04</td>
<td>96.8</td>
<td>9.6</td>
</tr>
<tr>
<td>yield (10% frequency, 10 calls of yield) / read sleep / write yield</td>
<td>t04</td>
<td>6.2</td>
<td>64.4</td>
</tr>
<tr>
<td>yield (10% frequency, 10 calls of yield) / read yield / write sleep</td>
<td>t04</td>
<td>94.4</td>
<td>7.2</td>
</tr>
</tbody>
</table>
Interesting Results for Assertion Errors

Even a very weak noise generated by the inserted code helps significantly
- The yield noise sometimes helps to achieve better results
- The sleep noise actually hides the errors back
- Using different noise injection settings for different types of memory accesses does not help much

<table>
<thead>
<tr>
<th>Noise configuration \ Program</th>
<th>t02</th>
<th>t12</th>
<th>t14</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>normal run</strong></td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td><strong>instrumented, no sleep or yield noise</strong></td>
<td>48.0</td>
<td>50.8</td>
<td>8.0</td>
</tr>
<tr>
<td>sleep (50% frequency, 10 ms of sleep)</td>
<td>0.0</td>
<td>0.0</td>
<td>1.2</td>
</tr>
<tr>
<td>yield (50% frequency, 10 calls of yield)</td>
<td>62.4</td>
<td>51.0</td>
<td>8.8</td>
</tr>
<tr>
<td>yield (50% frequency, 20 calls of yield)</td>
<td>64.6</td>
<td>55.2</td>
<td>6.6</td>
</tr>
<tr>
<td>yield (50% frequency, 10 calls of yield) / read 20 calls / write 5 calls</td>
<td>62.4</td>
<td>0.0</td>
<td>7.6</td>
</tr>
<tr>
<td>yield (50% frequency, 10 calls of yield) / read 5 calls / write 20 calls</td>
<td>64.0</td>
<td>0.0</td>
<td>10.4</td>
</tr>
<tr>
<td>yield (10% frequency, 10 calls of yield) / read sleep / write yield</td>
<td>60.6</td>
<td>0.0</td>
<td>9.4</td>
</tr>
<tr>
<td>yield (10% frequency, 10 calls of yield) / read yield / write sleep</td>
<td>47.4</td>
<td>0.0</td>
<td>3.4</td>
</tr>
</tbody>
</table>
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*
Plan of the talk

1. Introduction
2. Monitoring Multi-threaded C/C++ Programs
3. Fine-Grained Combinations of Noise
4. Experiments
5. Conclusion
Several problems which arise when monitoring C/C++ programs on the binary level were discussed.

Solutions to these problems were proposed.

An improvement of the noise injection technology was proposed.

The proposed solutions and improvements were validated on a set of C/C++ programs.
Future Work

- Support for other multi-threading libraries than pthreads
- Support for backtraces
- More experiments
- More sophisticated types of noises
- New detectors for concurrency errors
Related Work

IBM ConTest
- Only for Java, not freely available

ConTest for C
- Source level instrumentation
- Not supported anymore
- Not available for download

Fjalar
- Dynamic binary instrumentation
- Primarily designed to simplify access to compile-time and memory information
- Does not provide any concurrency-related information
End of presentation

Thank you for your attention!