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

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 naïvely, 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

Monitoring C/C++ Programs

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

Monitoring C/C++ Programs on the Binary Level

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.)

Monitoring Execution of Functions: A Problem

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    $0x602540,%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 unsychronised 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

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

Noise configuration \ Program	t01	t02
	% of err. runs	
instrumented, no sleep or yield noise	2.4	11.8
sleep (50% frequency, 10 ms of sleep)	69.2	46.6
sleep (10% frequency, 10 ms of sleep)	64.0	69.2
rs-sleep (50% frequency, 0–10 ms of sleep)	96.4	87.8
rs-sleep (10% frequency, 0–10 ms of sleep)	21.4	55.8
sleep (50% frequency, 10 ms of sleep) / read 20 ms / write 5 ms	64.8	89.4
sleep (10% frequency, 10 ms of sleep) / read sleep / write yield	34.2	81.0

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

Noise configuration \ Program	t06	t07
	% of err. runs	
instrumented, no sleep or yield noise	1.0	1.6
sleep (50% frequency, 10 ms of sleep)	53.6	69.4
sleep (10% frequency, 10 ms of sleep)	40.2	70.4
rs-sleep (50% frequency, 0–10 ms of sleep)	31.0	79.0
sleep (50% frequency, 10 ms of sleep) / read 5 ms / write 20 ms	92.6	96.2
yield (50% frequency, 10 calls of yield) / read yield / write sleep	95.0	99.6

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

Noise configuration \ Program	t04 (% of err. runs)
instrumented, no sleep or yield noise	1.2
sleep (50% frequency, 10 ms of sleep)	100.0
sleep (10% frequency, 10 ms of sleep)	56.0
rs-sleep (50% frequency 0–10 ms of sleep)	86.2
rs-sleep (10% frequency, 0–10 ms of sleep)	11.8
sleep (50% frequency, 10 ms of sleep) / read yield / write sleep	100.0
sleep (10% frequency, 10 ms of sleep) / read yield / write sleep	96.8

Interesting Results for Data Races

Different noise injection settings are sometimes the only thing that helps

% of err. runs

Noise configuration \ Program	t05
instrumented, no sleep or yield noise	0.0
sleep (50% frequency, 10 ms of sleep)	1.2
sleep (10% frequency, 10 ms of sleep)	5.4
rs-sleep (50% frequency, 0–10 ms of sleep)	0.6
rs-sleep (10% frequency, 0–10 ms of sleep)	0.0
sleep (50% frequency, 10 ms of sleep) / read 5 ms / write 20 ms	43.0
sleep (10% frequency, 10 ms of sleep) / read sleep / write yield	62.4

It is better to inject stronger noise before rarer accesses

% of err. runs

Noise configuration \ Program	t04	t05
instrumented, no sleep or yield noise	1.2	0.0
sleep (10% frequency, 10 ms of sleep) / read sleep / write yield	7.4	62.4
sleep (10% frequency, 10 ms of sleep) / read yield / write sleep	96.8	9.6
yield (10% frequency, 10 calls of yield) / read sleep / write yield	6.2	64.4
yield (10% frequency, 10 calls of yield) / read yield / write sleep	94.4	7.2

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

% of err. runs

Noise configuration \ Program	t02	t12	t14
normal run	0.0	0.0	0.0
instrumented, no sleep or yield noise	48.0	50.8	8.0
sleep (50% frequency, 10 ms of sleep)	0.0	0.0	1.2
yield (50% frequency, 10 calls of yield)	62.4	51.0	8.8
yield (50% frequency, 20 calls of yield)	64.6	55.2	6.6
yield (50% frequency, 10 calls of yield) / read 20 calls / write 5 calls	62.4	0.0	7.6
yield (50% frequency, 10 calls of yield) / read 5 calls / write 20 calls	64.0	0.0	10.4
yield (10% frequency, 10 calls of yield) / read sleep / write yield	60.6	0.0	9.4
yield (10% frequency, 10 calls of yield) / read yield / write sleep	47.4	0.0	3.4

Latest Results

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

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!