Fuzzing of Multithreaded Programs in .NET
Challenges and Solutions

http://d3s.mff.cuni.cz

Filip Kliber
kliber@d3s.mff.cuni.cz

CHARLES UNIVERSITY

faculty of mathematics and physics

Our Goal

- Implement a *fuzzer* for multithreaded C# programs
- That means a tool, capable of executing the program multiple times under different thread schedules
- In order to trigger some concurrency bugs

Department of Distributed and Dependable Systems D3S

Our Goal

- Implement a *fuzzer* for multithreaded C# programs
- That means a tool, capable of executing the program multiple times under different thread schedules
- In order to trigger some concurrency bugs

- First we need some terminology

Department of Distributed and Dependable Systems D3S

(brief) C# + .NET terminology

- C#: Programming Language
- .NET: Standard libraries + runtime environment
- CIL: The byte-code C# source is compiled to
- CLR: Virtual machine interpreting (or JIT compiling) CIL
- Managed code: Code executing within CLR
- Unmanaged code: Native code executing beyond the scope of CLR

(brief) .NET profiling API

- Microsoft provides a Profiling API for .NET applications
- The profiler (.dll) loads with CLR, and has callbacks for *interesting* events (method entry, thread creation, object allocation, ...)
- Doesn't include field accesses, but does work for properties (as they are translated to `get_*`, `set_*` methods)
- Utilized by writing a profiler in C++

Department of Distributed and Dependable Systems D3S

Our work — `ThreadingController`

- We used Profiling library to implement `ThreadingController`
- Directs the execution of multithreaded subject program (controls how is the execution of individual threads interleaved)
- By forcefully freezing/thawing the execution of OS thread in order to impose desired thread interleaving
- User defines a stop-point (a location in *source code*) that will freeze one/all threads when executed
- And a driver, which selects which thread (among all) should run

Department of Distributed and Dependable Systems D3S

`ThreadingController` algorithm

- C# is compiled into `.exe`
- CLR loads profiler and `.exe` and executes the byte-code
- When method call is encountered, the `ThreadingController` is notified
- If method name matches stop-point, thread(s) are frozen
- `ThreadingController` invokes the driver, which returns a (list of) thread(s) for thawing
- `ThreadingController` thaws selected thread(s) in specified order

Department of Distributed and Dependable Systems D3S

Drivers example

`console-driver`

- Prototype implementation that asks the user (on the fly) to select the thread to run
- Live example

Drivers (future work)

`fuzzing-driver`

- Different implementation of driver that selects thread to run **randomly** and provides a trace (log) of the program execution
- Next iteration will consult the trace to direct the execution in *different* manner
- Or reproduce a trace that was determined to be buggy (i.e. some assert failed)

Department of Distributed and Dependable Systems D3S

The encountered problems

- P#1: Forcefully stopping a thread of execution can cause problems
- P#2: Identifying the same thread across multiple executions of the subject program
- P#3: When to offer the driver to thaw some thread(s)

P#1 — Forcefully stopping the thread

- Action in one thread `T1` causes freezing of some other thread `T2`
- But `T2` is executing arbitrary code (even unmanaged code)

P#1 — Forcefully stopping the thread

- Action in one thread `T1` causes freezing of some other thread `T2`
- But `T2` is executing arbitrary code (even unmanaged code)

- Arbitrary = Scary
- Unmanaged = Dangerous

Department of Distributed and Dependable Systems D3S

P#1 — Cooperatively stopping the thread

- Action in one thread `T1` causes freezing of some other thread `T2`
- Halt the freeze, till `T2` is in managed context
- Managed = Safe

Department of Distributed and Dependable Systems D3S

P#1 — Cooperatively stopping the thread

- Action in one thread `T1` causes freezing of some other thread `T2`
- Halt the freeze, till `T2` is in managed context
- Managed = Safe

- But it can take arbitrary amount of time for the subject program to return from a function

Department of Distributed and Dependable Systems D3S

P#1 — Forcefully stopping the thread

- Action in one thread `T1` causes freezing of some other thread `T2`
- But `T2` is executing arbitrary code (even unmanaged)

- Arbitrary = Scary
- Unmanaged = Dangerous

P#1 — Forcefully stopping the thread

- Action in one thread `T1` causes freezing of some other thread `T2`
- But `T2` is executing arbitrary code (even unmanaged)

- Arbitrary = Scary
- Unmanaged = Dangerous

- How scary? What is the worst that could happen?

Department of Distributed and Dependable Systems D3S

P#1.1 — Heap allocation

- Heap allocation; `T2` stack can look like this:

```
7: _RtlEnterCriticalSection
6: __acrt_lock
5: heap_alloc_internal
4: malloc
...
2: Foo
1: Main
```

- P#1.1: profiler (driver) and subject program run in the same process
- Windows heap allocation algorithm locks the heap when allocating to prevent race condition
- While `T2` holds the lock, ALL heap allocations in the profiler are forbidden

Department of Distributed and Dependable Systems D3S

P#1.1 — Heap allocation

- Solution: Just don't use any heap allocation in the driver
- Most C++ containers (i.e. `std::vector`) need dynamic allocation to work
- Luckily, they can be parametrized with custom *`Allocator`*
- Custom allocator uses different heap (`CreateHeap`) to get the memory from (`HeapAlloc`)

P#1 — Forcefully stopping the thread

- Action in one thread `T1` causes freezing of some other thread `T2`
- But `T2` is executing arbitrary code (even unmanaged)

- Arbitrary = Scary
- Unmanaged = Dangerous

- How scary? What is the worst that could happen?

Department of Distributed and Dependable Systems D3S

P#1.2 — Deadlock with the profiler

- The profiler collects/caches lot of information
- CLR interacts with the profiler by invoking a callback from the thread that triggered the event
- Profiler will be invoked from different threads in parallel
- This requires some synchronization on the shared caches (i.e. mutex)
- Problem#1.2: `T2` can be frozen while holding this lock
- Shared caches can be in unspecified (broken) state

P#1.2 — Deadlock with the profiler

- Solution: Use separate (unmanaged) thread for all container modifying operations
- If a thread needs to update the cache, it writes data to predefined memory location via atomic Compare-And-Swap
- And waits (spin-lock) till the data is retrieved
- Dedicated thread updates the container
- And marks the operation as complete so that former thread can exit the spin-lock

P#1.2 — Deadlock with the profiler

- Note that using lock-free containers is not enough!
- Lock-free container can become frozen while updating it's internals
- (between atomic operations)
- Another thread trying to use the container would be spinning indefinitely
- Wait-free containers would work, but their functionality is limited

Department of Distributed and Dependable Systems D3S

The encountered problems

- P#1: Forcefully stopping a thread of execution can cause problems
- P#2: Identifying the same thread across multiple executions of the subject program
- P#3: When to offer the driver to thaw some thread(s)

Department of Distributed and Dependable Systems D3S

P#2 — Identifying same threads

- Identifying same threads across multiple executions is crucial to be able to reproduce a trace
- Thread of execution has several identifications in the profiler:
  - OS (`DWORD` via `GetCurrentThreadId`)
  - .NET Profiling API (`ThreadID` via callback parameters)
  - C++ (`std::thread::id` via `std::this_thread::get_id`)

Department of Distributed and Dependable Systems D3S

P#2 — Identifying same threads

- Identifying same threads across multiple executions is crucial to be able to reproduce a trace
- Thread of execution has several identifications in the profiler:
  - OS (`DWORD` via `GetCurrentThreadId`)
  - .NET Profiling API (`ThreadID` via callback parameters)
  - C++ (`std::thread::id` via `std::this_thread::get_id`)
- And none of those persist through application restart

Department of Distributed and Dependable Systems D3S

P#2 — Identifying same threads

- Solution: Introduce yet another thread id
- An `int` is incremented and assigned to every thread at creation
- I.e. first (Main) thread has id 1
- First user created thread will have id 2, then id 3, ...
- This can still be inconsistent as CLR informs the profiler about all events (including thread creation) in parallel, but testing shows it doesn't happen

- This only works if threads are created in consistent manner, i.e. doesn't work for thread pools, `Tasks`, data driven algorithms (i.e. sorting)

The encountered problems

- P#1: Forcefully stopping a thread of execution can cause problems
- P#2: Identifying the same thread across multiple executions of the subject program
- P#3: When to offer the driver to thaw some thread(s)

Department of Distributed and Dependable Systems D3S

P#3 — When to offer thawing

- Offering a thaw option right after threads are being frozen is natural
- Problem is thawing may not have any observable effect
- e.g. when a thread was frozen during waiting operation (like `Thread.Join`)
- Offering a thaw just once would deadlock the program (as all threads are either frozen or waiting)
- it's not possible to know if a running but waiting thread will progress (`Thread.Sleep`), or not (`Thread.Join`)

P#3 — When to offer thawing

- Solution: Offer thawing option multiple times (e.g. time based)
- This is reliable, but makes recreating the trace much more difficult
- as the trace is based on timing of actions
- We're still testing different approaches

Department of Distributed and Dependable Systems D3S

Current/Future work

- Implement exploring different traces
- Analyze reliability of different approaches for freezing/thawing in order to create new/recreate old trace

Department of Distributed and Dependable Systems D3S

Thank you

Questions?

Department of Distributed and Dependable Systems