SharpDetect: Dynamic Analysis Framework for C#/.NET Programs

Andrej Čižmárik and Pavel Parízek

Department of Distributed and Dependable Systems,
Faculty of Mathematics and Physics, Charles University,
Prague, Czech Republic
cizmarik.andrej@gmail.com, parizek@d3s.mff.cuni.cz

Abstract. Dynamic analysis is a popular approach to detecting possible runtime errors in software and for monitoring program behavior, which is based on precise inspection of a single execution trace. It has already proved to be useful especially in the case of multithreaded programs and concurrency errors, such as race conditions. Nevertheless, usage of dynamic analysis requires good tool support, e.g. for program code instrumentation and recording important events. While there exist several dynamic analysis frameworks for Java and C/C++ programs, including RoadRunner, DiSL and Valgrind, we were not aware of any framework targeting the C# language and the .NET platform. Therefore, we present SharpDetect, a new framework for dynamic analysis of .NET programs — that is, however, focused mainly on programs compiled from the source code written in C#. We describe the overall architecture of SharpDetect, the main analysis procedure, selected interesting technical details, its basic usage via command-line, configuration options, and the interface for custom analysis plugins. In addition, we discuss performance overhead of SharpDetect based on experiments with small benchmarks, and demonstrate its practical usefulness through a case study that involves application on NetMQ, a C# implementation of the ZeroMQ messaging middleware, where SharpDetect found one real concurrency error.

1 Introduction

Dynamic analysis is a popular approach to detecting possible runtime errors in software and for monitoring program behavior, which is applied within the scope of testing and debugging phases of software development. A typical dynamic analysis tool, such as Valgrind [5], records certain events and runtime values of program variables during execution of a subject program, and based on this information it can very precisely analyze behavior of the given program on the particular observed execution trace (and on few other closely-related traces). For example, dynamic bug detectors usually look for suspicious event sequences in the observed trace. Usage of dynamic analysis has already showed as beneficial especially in the case of multithreaded programs and search for concurrency errors, such as race conditions and deadlocks (cf. [2] and [6]), where the reported errors and fragments of the execution trace can be further inspected offline.

The main benefits of dynamic analysis include a very high precision and therefore also minimal number of reported false warnings, all of that because the actual concrete

program execution is observed. On the other hand, usage of dynamic analysis requires good tool support, which is needed for tasks such as program code instrumentation and processing of recorded important events. Tools should also have practical overhead with respect to performance and memory consumption.

While robust dynamic analysis frameworks have been created for Java and C/C++ programs, including RoadRunner [3], DiSL [4], Valgrind [5] and ThreadSanitizer [7], we were not aware of any framework targeting programs written in C# and running on the .NET platform. For that reason, we have developed SharpDetect, a framework for dynamic analysis of .NET programs, that we present in this paper.

SharpDetect takes executable .NET assemblies as input, performs offline instrumentation of the CIL (*Common Intermediate Language*) binary intermediate code with API calls that record information about program behavior, and runs the actual dynamic analysis of the instrumented subject program according to user configuration. Although the .NET platform supports many different programming languages, when developing and testing SharpDetect we focused mainly on programs compiled from the source code written in C#. Still, most programs written in other popular .NET languages, such as VB.NET and F#, should be also handled without any problems because SharpDetect manipulates the intermediate CIL binary code, but we have not tested it on any VB.NET and F# programs. In particular, the F# compiler may generate CIL code fragments different from those produced by C# compilers. We also want to emphasize that SharpDetect targets the modern cross-platform and open-source implementation of the .NET platform, which is called .NET Core. It runs on all major operating systems that are supported by .NET Core, that means recent distributions of Windows, Linux and Mac OS X. The output of SharpDetect includes a log of recorded events and a report of possibly discovered errors. Note, however, that SharpDetect is primarily a framework responsible for the dynamic analysis infrastructure. Specific custom analyses, including bug detectors, are actually performed by plugins that are built on top of the core framework. In order to demonstrate that the core framework (and its plugin API) is mature and can be used in practice, we have implemented two well-known algorithms for detecting concurrency errors, Eraser [6] and FastTrack [2], as plugins for SharpDetect. We have used the Eraser plugin in a case study that involves the NetMQ messaging middleware. Nevertheless, despite our focus on analyses related to concurrency, SharpDetect is a general framework that supports many different kinds of dynamic analyses.

Contribution and Outline. The main contributions presented in this paper include:

- SharpDetect, a new general and highly extensible framework that enables dynamic analysis of .NET programs;
- evaluation of runtime performance overhead incurred by usage of SharpDetect based on experiments with several benchmark programs written in C#;
- realistic case study that involves NetMQ, a C# implementation of the ZeroMQ messaging middleware, and demonstrates practical usefulness of SharpDetect for the purpose of detecting real concurrency errors.

The source code of a stable release of SharpDetect, together with example programs, is available at `https://gitlab.com/acizmarik/sharpdetect-1.0`.

Due to limited space, we provide only selected information about SharpDetect in this paper. Additional details can be found in the master thesis of the first author [1].

Structure. The rest of this paper is organized as follows. We describe the overall architecture and main workflow of SharpDetect in Section 2. Then we provide a brief user guide (Section 3), discuss the case study involving NetMQ (Section 4) and results of performance evaluation (Section 5), and finish with an outline of current work in progress and plans for the future.

2 Architecture and Main Workflow

SharpDetect consists of two parts, compile-time modules and runtime modules, that also correspond to main phases of its workflow, namely offline instrumentation and run of the dynamic analysis. The compile-time modules, Console and Injector, are responsible mainly for the offline CIL instrumentation. The runtime modules, Core and Plugins, perform the actual dynamic analysis during execution of the subject program. In addition, the module Common implements basic functionality, including the definitions of analysis events and necessary data structures, that is used by all other modules. Figure 1 shows a high-level overview of the architecture and workflow of SharpDetect. Both compile-time modules are displayed in the left frame with the label "Instrumentation", while runtime modules are displayed in the right frame with the label "Output Program". A very important aspect of the architecture of SharpDetect is that dynamic analysis runs in the same process as the subject program, in such a way that both our tool and the subject program share their memory address spaces. Now we provide details about individual modules and phases of the whole process.

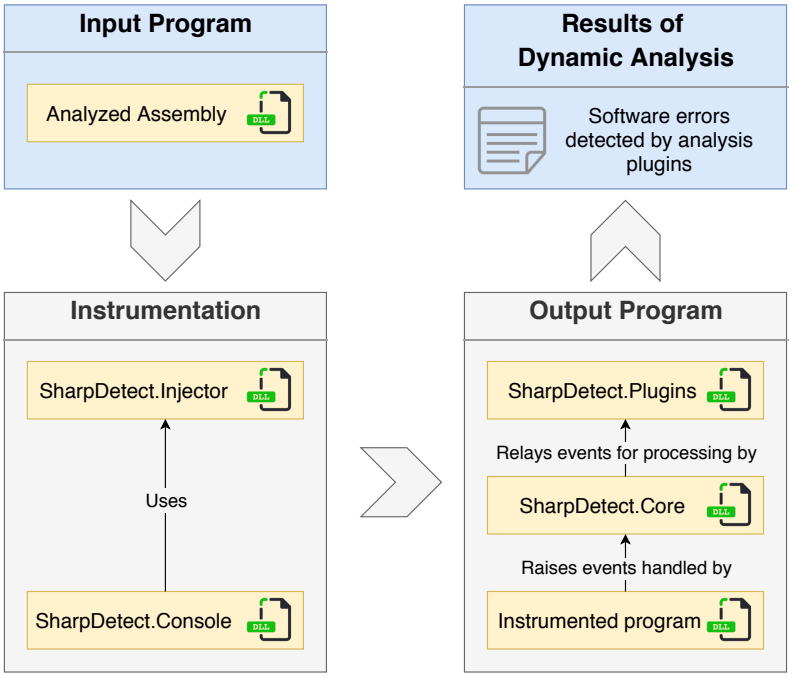

Fig. 1. High-level architecture and workflow of SharpDetect

The Console module is a .NET Core frontend of SharpDetect that has the form of a console application. It parses all configuration files and the command-line (Section 3), reads the input C# project and creates a self-contained package that includes the .NET program to be analyzed, drives the offline instrumentation process, and finally executes the actual dynamic analysis using the instrumented assemblies.

The Injector module uses dnlib [8] for manipulation with CIL bytecode. Its main purpose is to instrument the subject program with new code (classes, methods, and fields) that records the relevant events and other information about program state through calls of the SharpDetect API, when the dynamic analysis is executing. Note that SharpDetect instruments also .NET System libraries, especially the Base Class Library (BCL). This is needed, for example, to observe usage of collections in the subject program.

During the first phase, SharpDetect also completely removes native code from all the processed assemblies to enforce that CLR (Common Language Runtime), the virtual machine of .NET Core, actually loads the instrumented CIL bytecode instead of native images produced by the C# compiler based on the original CIL bytecode.

The Core module is the main component of SharpDetect that is used at runtime. It is responsible mainly for registering event handlers and dispatching of recorded analysis events to plugins. Like in the case of some other dynamic analysis tools, the list of supported events includes: accesses to fields of heap objects, accesses to array elements, dynamic allocation of new heap objects, method calls (processing invocation and return separately), thread synchronization actions (e.g., lock acquire, lock release, wait and notify signals), start of a new thread, and termination of a thread (e.g., via the join operation). Information about events of all these kinds is needed especially to enable detection of concurrency errors. Figure 2 illustrates the main event processing loop on an example involving the CIL instruction newobj for dynamic object allocation.

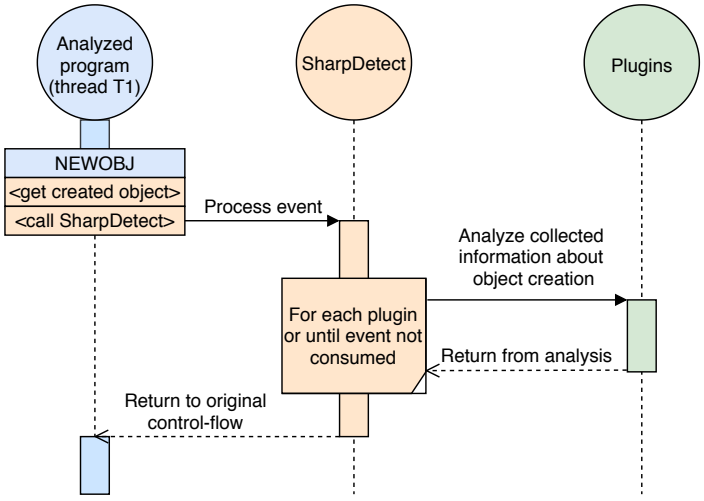

Fig. 2. Main event-processing loop illustrated on the CIL instruction newobj

The last part is the plugin API, an interface through which the core notifies plugins about observed events. Developers of custom plugins have to be aware of the fact that, due to SharpDetect running in the same process as the subject program, analysis of each individual event is carried out by the same thread that raised it. For each recorded event, the dynamic analysis engine takes control of the corresponding thread in the subject program for the duration of event's processing by all plugins. We provide more details about the plugin API from the user's perspective in Section 3.

A closely related aspect, which is not specific just to SharpDetect, is that when the analyzed program uses multiple threads, event handlers may be invoked concurrently and, therefore, events may be received in a wrong order by the analysis plugins. Consider the following example. Thread T_1 releases a lock L at some point during the program execution. But right before SharpDetect core notifies plugins about the corresponding event, a thread preemption happens and thread T_2 now runs instead of T_1. Immediately after the preemption, T_2 takes the lock L and SharpDetect notifies all plugins about this event. Plugins then receive information about T_2 acquiring L before the notification from T_1 about the release of L. We plan to address this challenge in future, using an approach that we discuss in Section 6.

One important limitation of the current version of SharpDetect is that it can track only information about user-defined threads. Specifically, it does not track analysis events related to threads retrieved from thread pools, because almost no information about such threads is available from the managed C# code.

3 User Guide

SharpDetect currently provides only a simple command-line interface. Figure 3 shows all three commands that must be executed in order to analyze a given C# program. The symbol rid in the first command stands for a runtime identifier, which needs to be specified in order to create a fully self-contained package. A complete list of supported runtime identifiers is provided in the official documentation for .NET Core [12].

```
// [optional] build the C# project and prepare it for
    instrumentation by generating a self-contained package
dotnet SharpDetect.Console.dll build <path_to_csproj> \
        --rid <platform_rid> --output <output_folder>

// instrument target assemblies based on the configuration
dotnet SharpDetect.Console.dll instrument <path_to_config>

// run the dynamic analysis
dotnet SharpDetect.Console.dll run \
        <path_to_instrumented_assembly> \
        --config <plugins_registration>
```

Fig. 3. Example usage of SharpDetect through its command-line interface

Configuration. Before the subject program can be analyzed, the user has to prepare the configuration of SharpDetect. In a local configuration file, specific to a given pro-

gram, the user can (1) disable some categories of analysis events and (2) further restrict the set of reported events by defining patterns for names of methods and object fields that should be tracked. Figure 4 shows an example configuration, written in the JSON syntax, that:

- enables analysis events related to field accesses, method calls, and object allocation;
- completely disables all events related to arrays;
- restricts the set of reported field access events for the assembly MyAssembly1.dll just to the class C1 in the namespace nsA;
- and finally restricts the set of reported method call events for the assembly just to the class C1 in the namespace nsA and the method Mth3 in the class C2 from the namespace nsB.

The main purpose of all these configuration options is to allow users to specify events relevant for a particular run of dynamic analysis, so that the overall number of reported events is significantly reduced and the output can be therefore more easily inspected.

```
{
  "TargetAssembly" : "MyAssembly1.dll",
  "FieldPatterns" : [ "nsA.C1" ],
  "MethodPatterns" : [ "nsA.C1", "nsB.C2::Mth3" ],

  "FieldInjectors" : true,
  "MethodInjectors" : true,
  "ObjectCreateInjector" : true,
  "ArrayInjectors" : false
}
```

Fig. 4. Example content of a local configuration for a specific dynamic analysis

Users also need to decide upfront whether they want to enable JIT optimizations. The difference can be observed, for example, in the case of programs that use multiple threads or the Task Parallel Library (TPL) with one simple lambda function as a task body. If the JIT optimizations are enabled, then each execution of the lambda function might be performed by the same thread, regardless of the usage of TPL. On the other hand, when the JIT optimizations are disabled, each execution of the lambda function is performed by a different thread.

Plugins. We have already indicated that a very important feature of SharpDetect is the possibility to use custom analysis plugins. Developers of such plugins need to implement the abstract class BasePlugin that belongs to the Core module. In Figure 5, we show those methods of the abstract class that correspond to the most commonly used analysis events. Signatures of the remaining methods follow the same design pattern. The full source code of the BasePlugin class is available in the project repository and it is also documented on the project web site.

The command that executes actual dynamic analysis (Figure 3) takes as one parameter the list of plugin names in the format plugin1 | plugin2 | ... | pluginN. SharpDetect then looks for available plugins in the directory specified by the environment variable

```
public abstract string PluginName { get; }
void AnalysisStart(MethodDescriptor entryMethod);
void AnalysisEnd(MethodDescriptor entryMethod);
void FieldRead(int threadId, object obj, FieldDescriptor fd);
void FieldWritten(int threadId, object obj, FieldDescriptor fd,
     object newValue);
void LockAcquireAttempted(int threadId, MethodDescriptor mth,
    object lockObj, (int, object)[] parameters);
void LockAcquireReturned(int threadId, MethodDescriptor mth,
    object lockObj, bool result, (int, object)[] parameters);
void LockReleased(int threadId, MethodDescriptor mth, object
    lockObj);
void MethodCalled(int threadId, (int, object)[] parameters,
    MethodDescriptor mth);
void MethodReturned(int threadId, object retValue, bool valid,
    (int, object)[] parameters, MethodDescriptor mth);
void ObjectCreated(int threadId, object obj);
void UserThreadStarted(int threadId, Thread thread);
void UserThreadJoined(int threadId, Thread thread);
```

Fig. 5. Selected methods defined by the abstract class BasePlugin

SHARPDETECT_PLUGINS. During the analysis run, every observed event is dispatched by SharpDetect to the first plugin in the chain. A plugin that received an event may consume the event or forward it to the next plugin. Note that the default implementation of all event handler methods on the abstract class BasePlugin forwards the information about events to the next plugin in the chain, if such plugin exists.

Additional technical details regarding the development of custom plugins are illustrated by two example plugins that we released together with SharpDetect, i.e. our implementations of the algorithms Eraser and FastTrack that can be found in the module SharpDetect.Plugins.

4 Case Study

We have applied SharpDetect to the NetMQ library [9], which is a C# implementation of the ZeroMQ high-performance asynchronous messaging middleware, in order to see how well it can help with debugging of concurrency issues in realistic programs. To be more specific, the first author used SharpDetect when searching for the root cause of a particular timing issue in NetMQ that occurred very rarely.

The source code of a test program that uses NetMQ, together with the configuration of SharpDetect, is in the directory src/SharpDetect/SharpDetect.Examples/CaseStudy of the repository at `https://gitlab.com/acizmarik/sharpdetect-1.0`. It is a standard .NET Core console application that runs two threads (server and client). Here we describe just a fragment of the SharpDetect's output and a fragment of the NetMQ source code that contains the root cause of this particular concurrency issue.

Figure 6 shows output produced by the run of dynamic analysis with the Eraser plugin, which can detect possible data races. The last two entries in Figure 6 represent the warning reported by Eraser, which points to possible data races on the static

```
// Field s_lastTime was written by thread with ID=3
21:12:57 [INF] [3] Field: System.Int64 NetMQ.Core.Utils.Clock::
    s_lastTime was written with value 27266154.

// Field s_lastTsc was read by thread with ID=4
21:12:57 [INF] [4] Field: System.Int64 NetMQ.Core.Utils.Clock::
    s_lastTsc was read from.

// Field s_lastTime was read by thread with ID=3
21:12:57 [INF] [3] Field: System.Int64 NetMQ.Core.Utils.Clock::
    s_lastTime was read from.

// Field s_lastTsc was written by thread with ID=4
21:12:57 [INF] [4] Field: System.Int64 NetMQ.Core.Utils.Clock::
    s_lastTsc was written with value 54333378824957.

21:12:57 [ERR] [Eraser] detected data-race on a static field
    System.Int64 NetMQ.Core.Utils.Clock::s_lastTsc
21:12:57 [ERR] [Eraser] detected data-race on a static field
    System.Int64 NetMQ.Core.Utils.Clock::s_lastTime
```

Fig. 6. Output produced by SharpDetect with the Eraser plugin for the program that uses NetMQ

fields s_lastTsc and s_lastTime defined by the class NetMQ.Core.Utils.Clock. In the corresponding revision of NetMQ [10], both static fields are read and written only by the method NowMS whose source code is displayed in Figure 7. Log entries at the level INF, which are presented in Figure 6, indicate that the method NowMS is actually executed by multiple threads without any synchronization of the critical section.

5 Performance Evaluation

In this section we report and discuss the overhead of dynamic analysis with SharpDetect, and the impact on analyzed programs, in terms of the running time and memory consumption. For that purpose, we performed experiments with two small benchmark programs on the following hardware and software configuration: Intel Core i7-8550U CPU with the clock speed 1.80 GHz and 4 cores, 16 GB of memory, 64-bit version of Windows 10, and .NET Core 2.1.

The first benchmark program uses Task Parallel Library (TPL) to process a big array in an unsafe way, such that individual threads are not synchronized and therefore a data race may happen at each access to array elements. The second benchmark program is a simple implementation of the producer-consumer pattern, where (i) both the producer and consumer are implemented as separate Task objects that share a common queue and (ii) access to the queue is guarded by a lock. Source code of both programs is available in the SharpDetect repository in the directories src/SharpDetect/SharpDetect.Examples/(Evaluation1,Evaluation2). Even though these programs are quite small, their execution generates a lot of analysis events, which makes them useful for the purpose of measuring the overhead of analysis with SharpDetect.

```
public static long NowMs() {
  long tsc = Rdtsc();
  if (tsc == 0) return NowUs() / 1000;

  /* Beginning of critical section */
  if (tsc - s_lastTsc <= Config.ClockPrecision / 2
        && tsc >= s_lastTsc) {
    return s_lastTime;
  }

  s_lastTsc = tsc;
  s_lastTime = NowUs() / 1000;
  return s_lastTime;
  /* End of critical section */
}
```

Fig. 7. Source code of the method NowMS that contains root cause of the concurrency issue

Each measurement was repeated 50 times. In tables with results, we present average values together with the corresponding standard deviations. The baseline values of running time and memory consumption, respectively, were recorded using the subject programs before instrumentation.

Table 1 contains results for the first benchmark and several configurations of SharpDetect. Data in the table show that usage of SharpDetect, together with the Eraser plugin, is responsible for a slow-down by the factor of 4 with respect to the baseline. Memory overhead is caused by tracking analysis information for each array element.

Configuration		Results	
Instrumented	**Plugins**	**Time (s)**	**Memory (KiB)**
No (baseline)	–	0.19 ± 0.01	333
Yes	EmptyPlugin	0.44 ± 0.01	541 ± 5
Yes	FastTrack	0.56 ± 0.01	4223 ± 37
Yes	Eraser	0.79 ± 0.03	7216 ± 6

Table 1. The running time and memory consumption of the first benchmark program

Table 2 contains results for the second benchmark. In this case, we observed slow-down at most by the factor of 3.7. Memory consumption apparently increased by the factor of 16, but, in fact, there is a constant memory overhead of about 4000 KiB, regardless of the configuration. The main cause is that SharpDetect needs to track a lot of information during the program execution, such as method call arguments and return values, even when plugins do not use much of the data.

Overall, results of our experiments indicate that SharpDetect has a relatively small overhead that enables usage of the tool in practice. Note that baseline measurements

Configuration		Results	
Instrumented	**Plugins**	**Time (s)**	**Memory (KiB)**
No (baseline)	–	0.145 ± 0.003	261.1
Yes	EmptyPlugin	0.51 ± 0.01	4265.7 ± 0.5
Yes	Eraser	0.53 ± 0.02	4267.5 ± 0.5
Yes	FastTrack	0.54 ± 0.02	4268.3 ± 0.7

Table 2. The running time and memory consumption of the second benchmark program

of the memory consumption did not deviate at all for both programs, because non-instrumented variants of the programs allocate very few objects on the heap.

6 Future Work

We plan to continue our work on SharpDetect in various directions, implementing new features and improving its performance.

One way to reduce the effects of dynamic analysis with SharpDetect on the behavior and performance of subject programs is to use the .NET Profiling API [11], which enables online instrumentation at the level of CIL bytecode during execution of a subject program. In addition, usage of the .NET Profiling API allows clients to observe specific events raised by the .NET execution engine, CoreCLR, even without code instrumentation. We are currently working on the implementation of a new version, SharpDetect 2.0, that will (1) utilize the .NET Profiling API, (2) execute dynamic analysis using the out-of-process approach where the analysis runs in a different process than the subject program, and (3) contain many additional improvements. The process of the subject program will contain just the minimal necessary amount of injected code to record events and forward them to the analysis process (which involves all the plugins, too).

Another goal is to address the issues related to possible concurrent invocation of event handlers that we described at the end of Section 2. We plan to implement a solution that will impose more strict ordering of analysis events from multiple threads, based on some form of vector clock, such that SharpDetect could delay dispatching of an event until all observed preceding events are processed.

Acknowledgments. This work was partially supported by the Czech Science Foundation project 18-17403S and partially supported by the Charles University institutional funding project SVV 260588.

References

1. Andrej Cizmarik. Dynamic Analysis Framework for C#/.NET Programs. Master thesis, Charles University, Prague, February 2020. https://is.cuni.cz/webapps/zzp/detail/209472/45410198
2. Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In Proceedings of PLDI 2009, ACM. https://doi.org/10.1145/1542476.1542490

3. Cormac Flanagan and Stephen N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In Proceedings of PASTE 2010, ACM. https://doi.org/10.1145/1806672.1806674
4. Lukas Marek, Yudi Zheng, Danilo Ansaloni, Aibek Sarimbekov, Walter Binder, Petr Tuma, and Zhengwei Qi. Java Bytecode Instrumentation Made Easy: The DiSL Framework for Dynamic Program Analysis. In Proceedings of APLAS 2012, LNCS 7705. https://doi.org/10.1007/978-3-642-35182-2_18
5. Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In Proceedings of PLDI 2007, ACM. https://doi.org/10.1145/1250734.1250746
6. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems (TOCS), 15(4), 1997, ACM. https://doi.org/10.1145/265924.265927
7. Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In Proceedings of WBIA 2009, ACM. https://doi.org/10.1145/1791194.1791203
8. Dnlib. https://github.com/0xd4d/dnlib (accessed in June 2020)
9. The NetMQ library. https://netmq.readthedocs.io/en/latest/ (accessed in June 2020)
10. NetMQ Custom Clock Implementation. https://github.com/zeromq/netmq/blob/e4dfcf9e8190f85bf4fab9fc657e2c7da820c7f4/src/NetMQ/Core/Utils/Clock.cs#L88 (accessed in June 2020)
11. .NET Profiling API Reference. https://docs.microsoft.com/en-us/dotnet/framework/unmanaged-api/profiling/ (accessed in June 2020)
12. A complete list of supported Runtime Identifiers (RID). https://docs.microsoft.com/en-us/dotnet/core/rid-catalog (accessed in June 2020)