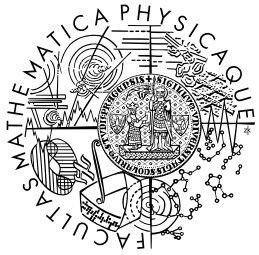

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Andrej Čižmárik

Dynamic Analysis Framework for C#/.NET Programs

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Pavel Parízek, Ph.D.
Study programme: Computer Science
Study branch: Software Systems

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor, Pavel Parízek, for his guidance, valuable advice, and his genuine interest both in the implementation part, as well as in this thesis. I am also very grateful for the support and encouragement that my family and friends provided throughout my whole master's studies.

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

Author: Andrej Čižmárik

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Dynamic analysis is a technique used to analyse the behaviour of programs, which can be utilized when searching for various software errors. Nowadays, there is a trend in software development towards multi-threaded programs that are, undeniably, prone to race conditions. Furthermore, software errors that stem from timing issues and incorrect ordering of operations across individual threads are generally hard to find, since they are by nature non-deterministic. We decided to implement a dynamic analysis framework for C# programs, along with two well-known algorithms capable of detecting and predicting data-races. As a result, we created an extensible and configurable tool, SharpDetect, that supports dynamic analysis of CIL programs created by compilation from C# source code on platforms supported by .NET Core. To demonstrate its practical usefulness, SharpDetect was successfully applied on NetMQ, C# implementation of ZeroMQ, where it found one real software error.

Keywords: dynamic analysis, data-races, .NET Core

Contents

1. Introduction

Software errors, more commonly referred to as ”bugs”, have been troubling developers throughout the history of software engineering. Their consequences can be, however, catastrophic – some well-known software errors were linked with disasters, such as, Ariane 5 explosion shortly after launch [1] or Therac-25 radiation therapy machine delivering lethal radiation doses [2].

One of the reasons for occurrence of many errors is complexity. Undoubtedly, some problems are hard to solve by their nature and therefore require proper design and implementation. However, due to ever-changing requirements, growing code-base and improper design, software can easily become hard to maintain. As a result, this may clearly lead to more software errors. Moreover, even in well-maintained software products, errors can be hard to find and replicate. Then after successful replication it can be also significantly hard to remove them.

Nowadays we can observe a trend in development of concurrent and parallel programs. This may be driven by the fact that modern CPUs have not been getting significantly faster per a single core as they used to in the past, but instead they tend to have multiple cores that need to be utilized in order to gain the best performance. Furthermore, concurrency brings another source of possible software errors which are by nature non-deterministic and therefore harder to even observe. Development of parallel programs is also considered more complex as it affects normal execution flow. As a result, developers need to ensure that programs behave correctly after every possible interleaving of machine instructions. This is generally the place were developers tend to make mistakes when working on concurrent programs.

Due to the mentioned reasons, there have been a major effort to create tools that could simplify the process of finding and replicating software errors. As a result, a variety of tools and techniques were adopted – from creating and analysing memory dumps, through using debuggers and profilers to utilizing tools that analyse the behaviour of programs and verify specified properties. Individual tools and techniques are better suited for different problems, however, they all contribute to easier, more manageable and also more dependable software development process.

One of the techniques that can be used to search for different kinds of software errors is dynamic analysis. The aim of this technique is to analyse the behaviour of a program based on a single execution path. Information about the execution path is usually obtained by instrumenting analysed executables to raise events on, for example method calls and memory accesses. Dynamic analysis has been already successfully deployed even in bigger solutions, for instance by Facebook [3].

Even though dynamic analysis can be useful when searching for software errors, there are tools available only for some programming languages (executable types). To name a few, there is a tool called Valgrind [4] for various hardware platforms and operating systems or a tool called RoadRunner [5] for java bytecode. Mentioned tools provide users with the ability to search for various kinds of software errors and even implement their own plugins and

analysers on top of them. This is certainly something which would be beneficial also for other platforms when searching for software errors.

One of such platforms is .NET which became a popular choice for writing software that is not only easy to write, memory and type safe, but also cross-platform. While the platform has currently multiple implementations, the most important one from the long point of view seems to be .NET Core – the successor of the previously predominant implementation called .NET Framework [6]. Despite the recent popularity of .NET, at the time of writing this thesis, there are no freely available tools for dynamic analysis of .NET programs.

1.1 Project Goals and Contribution

The ultimate ambition of this project is to design and implement a dynamic analysis framework for the .NET platform. As we previously mentioned, we aim to support modern cross-platform and open-source implementation called .NET Core. In order to achieve this, we will present several major goals which we aim to fulfil while working on this project.

(G1) Extensibility

Users of the framework should be able to implement their own analysis extensions. This can be achieved by providing a pluggable environment, simple API and easy-to-follow guidelines for successful implementation and integration of custom extensions.

(G2) Configurability

Dynamic analysis can have significant performance overhead on the analysed programs. To counter this issue, users should be allowed to specify what aspects of the program behaviour should be analysed and what should be skipped. The main purpose of this goal is to reduce events from unrelated parts of programs, as well as filter whole event categories.

(G3) Performance

The execution of analysed programs should not be performance-wise affected more than necessary. While dynamic analysis is known to negatively affect performance of programs, we should aim to minimize this overhead when designing and implementing the framework.

(G4) Support for multiple platforms

As we already mentioned, .NET Core supports multiple platforms – Windows, Linux and OS X. Since developers can use .NET Core on multiple platforms, our dynamic analysis framework should support an additional platform apart from Windows as well.

As a result of the mentioned goals above, we designed and implemented a dynamic analysis framework called **SharpDetect**. An important part of

SharpDetect are also two extensions that implement well-known algorithms, specifically Eraser [7] and FastTrack [8], which can be used to discover and predict concurrency issues in analysed programs. We also prepared several examples and illustrative tutorials on how to use SharpDetect to analyse the behaviour of programs.

1.2 Thesis Outline

The rest of this thesis is organized as follows; we start the next chapter by further describing dynamic analysis, our chosen approach for analysing the behaviour of programs. Most importantly, we will discuss its advantages over other techniques, as well as its drawbacks and how we may try to counter them. We finish the chapter by comparing various approaches on how we can capture dynamic analysis events using available tools for .NET platform.

In the third chapter, we will look at the Common Language Infrastructure, which is a specification of the .NET environment. We will be mostly interested in the description of its executable code and runtime. After we introduce general concepts, we will continue by describing some implementation-specific details of the selected framework – .NET Core. This will help the reader understand from a low-level point of view how we can implement dynamic analysis for the .NET platform.

The main focus of the fourth chapter will be the design and implementation of the created dynamic analysis tool, SharpDetect. Apart from the architectural overview and the description of its implementation, we will organize this chapter as a development diary. While working on SharpDetect, we faced many interesting problems that we tried to solve in various ways. In this chapter, we will discuss the solutions we tried, why some of them did not work, as well as, how we finally solved the respective problems. At the end of the chapter, we describe possible improvements that are left as future work.

The fifth chapter will be dedicated to the evaluation of our project. We will discuss implemented performance optimizations, as well as, present measurements on how SharpDetect impacts the performance of selected subject programs.

In the sixth chapter, we will present tutorials and more in-depth examples on how to use SharpDetect to analyse .NET programs. Moreover, we discuss guidelines for implementing analysis plugins and describe some interesting aspects of the already implemented plugins.

The whole seventh chapter is dedicated to a case study where we apply SharpDetect on an existing .NET library in order to search for possible concurrency issues. We describe the process of preparing the subject program, configuring SharpDetect, and evaluating obtained results.

Then, in the last chapter, we will present the conclusion of the whole thesis. Furthermore, we will discuss how we managed to fulfil thesis goals that we presented in the previous section.

2. Dynamic Analysis

Dynamic analysis is a technique that can be used to analyse the behaviour of programs during their runtime based on a single execution path. This approach can be used to obtain truly precise information about the given execution path, considering the fact that its runtime behaviour is directly observed. The downside is that only a single execution path is analysed and therefore, the coverage of dynamic analysis depends on the quality of the provided test suite. However, we gain precision, as compared to static analysis that generally approximates obtained information about the program's behaviour because it tries to reason about it without actually executing the program.

In the following sections, we will thoroughly describe individual analysed events that might be interesting when analysing the behaviour of programs and possible ways how to observe and capture them. Then we continue by discussing available .NET tools that could help us implement a dynamic analysis framework for .NET Core. Lastly, we provide a comparison of the tools and we choose an approach, as well as, a tool that suite our use-case the most.

2.1 Analysed Events

In this section, we discuss individual events that can be observed during dynamic analysis and contribute to the comprehension of the analysed program's behaviour. Specifically, we are interested in local operations with the evaluation stack, such as load and store memory, dynamic objects allocation, as well as method calls. In fact, method calls alone cover also some special events, for example user-threads manipulation, synchronization and signals. Together, mentioned events provide a powerful information about the analysed program's behaviour and as such we can split them into the following categories:

- **Memory accesses**: volatile and non-volatile memory accesses performed on fields and array elements
- **Methods**: calls and returns of methods together with their arguments and return values
  - **Threads**: creating, starting and joining of user-created threads
  - **Locks**: acquire and release of locks on objects
  - **Signals**: wait and notify signals on objects
- **Objects**: dynamic allocations on the heap
- **Classes**: class constructors

As indicated, presented events can be already used to check many interesting properties of analysed programs, for example detection and prediction of possible concurrency issues, such as deadlocks, race conditions and atomicity violations.

2.2 Capturing Events

Now that we covered the basic set of events that we need to observe and capture information about, we can discuss how to achieve this. There are actually multiple ways to observe the behaviour of programs which will be further discussed in the following subsections.

2.2.1 Runtime Interpretation

In order to obtain required events, we could use a modified interpreter for .NET programs where some of its routines would simply notify our dynamic analysis framework. Since it is clear that interpreters have complete information about program execution, this approach would certainly work.

However, the development of such interpreter would be a significant task on its own. Even though the official .NET Core repository contains an interpreter [9], it is not production-ready and it is advised not to use it. In fact the only use-case for this interpreter is currently when porting .NET Core to a different platform where some of its components are not ready for the given platform yet. It is also noticeably slower compared to a normal execution.

2.2.2 Code Instrumentation

Code instrumentation works by modifying programs either on the source code level or on the bytecode level. This grants us the ability to insert custom code into individual methods before they get executed by runtime. Therefore, we could locate occurrences of analysis events and instrument them so that they capture information about the event and notify the analysis framework.

In our case, however, instrumentation on bytecode level is clearly preferred. For example, there is a bigger chance that language designers introduce new constructs, compared to when there is a need to change the underlying bytecode which happened in .NET so far only once [10]. Additionally, there are multiple languages that compile to the same bytecode, such as C#, F# or Visual Basic .NET. Therefore, when supporting instrumentation on the bytecode level, we can easily provide better coverage in terms of different languages, as well as language constructs.

Although we showed that bytecode instrumentation is the preferred approach, there are actually two options how to implement it. The first option works online during the analysed program's execution and the second option is offline and as such works ahead of time to the actual program's execution. In case of the online approach, we need further support from runtime – the ability to observe method calls, inspect their bytecode and make on-the-fly modifications. This is certainly more complicated compared to the offline approach where we can work with any bytecode inspector capable of generating code. From the execution point of view, the offline approach does not negatively affect start-up time of dynamic analysis as compared to the online approach. On the other hand, with the online approach everything can be easily performed in-memory without the need to store any generated code in persistent storage.

2.3 Available Tools for .NET

Since we already described available approaches on how to observe and capture information about the behaviour of programs, we need to further analyse available options for the implementation of these approaches. Therefore, the following subsections are dedicated to the analysis of available tools for .NET using which we can implement a dynamic analysis framework. Later, we will present the results – an approach and a tool we will use throughout the rest of this thesis.

2.3.1 Roslyn

Roslyn [11] is an open-source C# and Visual Basic .NET compiler with a powerful static code analysis API. The biggest advantage is Roslyn's philosophy which offers its capabilities in a *Compiler as a Service* fashion. This means that we can actually access results of individual stages of compilation, as well as modify them. Therefore, Roslyn would be a powerful candidate for instrumentation on the source code level.

However, we already mentioned the disadvantages in connection with this approach and here with Roslyn we can observe the same problem once again. Even though Roslyn is together with its code analysers very powerful, it is more suited for static analysis. Additionally, by using Roslyn we would miss on some .NET languages, for example, F# which has its own compiler infrastructure [12] that is independent from Roslyn.

2.3.2 Mono.Cecil

Mono.Cecil [13] is an open-source library for inspecting and generating code that supports .NET programs. The library provides powerful object representation of .NET programs and is widely used in many projects that need to inspect, manipulate or generate bytecode. Moreover, there are many available examples and tutorials together with a fairly active online community.

Despite the popularity and maturity of this project, it lacks some features that can be used by some .NET libraries. An example of such feature is the support for mixed-mode assemblies. Apart from bytecode, mixed-mode assemblies contain also native code that is specific for the underlying platform. Although most .NET libraries do not use such features, core libraries are known to use this mainly for performance benefits.

2.3.3 dnlib

There is another library that covers a similar use-case as Mono.Cecil, called dnlib [14]. The difference is that dnlib additionally supports, for example obfuscated .NET libraries and already mentioned mixed-mode assemblies.

Even though this library has not been around for so long as Mono.Cecil and provides significantly less examples, it covers more use cases. The ability to instrument also framework libraries would be very helpful later when implementing the dynamic analysis framework. Therefore, dnlib is in our case a better candidate for instrumentation on the bytecode level.

2.3.4 Profiling API

One of the key features of .NET runtime is that is provides a profiling API [15]. This API grants users the ability to observe program execution by raising various events whenever, for instance, a method is being compiled, a dynamic allocation is being handled or a thread is starting to run, and many more.

Despite the fact that this approach provides the functionality we require, it is an unmanaged API that is hard to use from high-level .NET languages. Furthermore, any calls to a .NET library within the profiling API can easily cause an endless recursion because its execution may end in a profiling routine, as well. Additionally, at the time of writing this thesis, there are no similar tools like Mono.Cecil or dnlib available for other platforms than .NET that would be usable in combination with the profiling API.

Compared to other presented tools, the profiling API is the only candidate that operates at the low-level – directly interacting with the .NET runtime. Moreover, at the time of writing this thesis, it is probably the only viable option for online bytecode instrumentation. However, this API is harder to use and there is only limited information on how use it with examples mostly available for older runtime used by the .NET Framework.

2.4 Summary

To sum up our analysis, we clearly showed that instrumentation is a better choice compared to interpretation, as there is no customizable and production-ready interpreter available for the .NET platform. Moreover, offline instrumentation would not provide any major disadvantages over the online approach. Therefore, we decided to continue our work on this thesis using the offline instrumentation approach with the help from dnlib library.

3. Common Language Infrastructure

The Common Language Infrastructure (CLI) is a specification standardized by ECMA 335 [16] that describes the virtual execution system of .NET programs and its executable code format. The CLI is platform-agnostic and supports high-level programming languages. Executable code then depends only on the target virtual execution system and does not need to be modified to execute on different platforms. In the following sections, we will further describe the CLI and provide details about some well-known implementations.

3.1 Common Language Runtime

The Common Language Runtime (CLR) is a CLI-compatible runtime environment. More specifically, the CLR is a high-level virtual machine whose main purpose is to make programming easier and support high-level languages. Some of its key features are the following (cf. The Book of the Runtime [17]):

- **Garbage collection** (GC) relieves programmers from the requirement to explicitly delete no longer needed memory. The CLR keeps track of all references to dynamically allocated objects on the garbage-collected heap. The objects are always guaranteed to be deleted by GC at a safe point of execution.

  Various CLI implementations additionally perform heap compaction, which is a way to minimize fragmentation. In order for the compaction to work, the CLR needs to move live objects and subsequently fix corresponding references in the program.

- **Memory safety** requires use of allocated memory only. Together with some runtime checks and prohibition of dangling pointers, the CLR guarantees that programs are memory safe. There are some exceptions, for example the usage of unsafe blocks.

- **Type safety** requires that each memory allocation is associated with a type. Moreover, for each memory allocation the program can perform only operations which are valid for the given type. Similarly to memory safety, unless the program is compiled as unsafe, it is guaranteed to be type safe.

- **Verifiable code** is both memory and type safe. Such code is guaranteed not to contain technical errors, for example segmentation faults, but only logical ones. Due to this fact, it is highly advised to use verifiable code whenever possible.

We can see from the mentioned features above that programming for the CLR can be safer and less error-prone compared to traditional languages like C and C++.

When programming for the CLR, the code is placed together with some metadata into modules. Individual modules are then packed in assemblies,

which have usually `.exe` or `.dll` extension. The code contained in the modules is called managed code. In the following section, we will further describe the difference between managed and unmanaged code. Then in the subsequent section, we will focus on the language that is used when programming for the CLR.

3.2 Managed and Unmanaged Code

In order for the CLR to be able to provide certain functionality, such as the already mentioned GC, it needs to have complete information about the running code. This information is either obtained from metadata embedded into modules during compilation, or it can be obtained by observing its execution. Code that provides this information for the CLR is called managed code.

On the other hand, unmanaged code, more commonly known as native code, generally does not provide this information and therefore cannot be further observed nor controlled by the CLR. Interoperability with unmanaged code is, however, necessary in order for managed code to communicate with the environment. As unmanaged code we can consider any code that started before the initialization of the CLR, which is on its own mostly composed from unmanaged code.

In the next section, we will focus on the description of the Common Intermediate Language (CIL), which is the language used when programming for the CLR and generates managed code. Later, in one of its subsections which is dedicated to method calls, we will also revisit unmanaged code, when discussing some of the techniques that can be used when calling into unmanaged code.

3.3 Common Intermediate Language

Common Intermediate Language (CIL) is an instruction set for an abstract object-oriented and stack-based virtual machine. Having an abstract machine as a compilation target, it is necessary for the code to be either interpreted or just-in-time (JIT) compiled into native code. By default, the CLR uses a JIT compiler.

An example of a code written in CIL can be seen in the following code-listing.

```
1 .method private hidebysig static void Main (string[] args)
2   cil managed {
3   .maxstack 8
4   .entrypoint
5
6   IL_0000: ldstr "Hello World!"
7   IL_0005: call void [System.Console]System.Console::WriteLine
        (string)
8   IL_000a: ret
9 }
```

Listing 3.1: Hello world written in CIL

First, the string `"Hello World!"` is pushed onto the evaluation stack. The string then gets popped by a call to a static method `WriteLine` defined by the

class `System.Console`. On return, `WriteLine` does not push anything on the evaluation stack, which is denoted by the `void` return type. Finally, the `Main` method returns.

In the following subsection, we will briefly introduce the Common Type System, which is necessary to understand basic concept of types in .NET and then in the subsequent subsection, we will focus on programming using CIL.

3.3.1 Common Type System

The Common Type System (CTS) standardizes how are type definitions and their corresponding runtime values represented in memory. This is helpful for supporting multiple languages, as well as interoperability between them [17]. Apart from this, CTS distinguishes three basic categories of types – pointer types, reference types and value types.

Pointer Types

In an unsafe block of code, programmers can use `C`-like raw pointers. Pointers are not derived from `System.Object` and cannot be converted to anything apart from different pointer types and integral types. Usage of this feature, however, always results in an unverifiable code as described in Section 3.1.

Reference Types

Reference types are always allocated on the heap. Therefore, variables of reference types always contain a reference to an object stored on the heap. Additionally, when passing reference types as method arguments, their reference is always passed by copy, unless specified otherwise. Passing by reference semantics can be achieved using the `in`, `ref` or `out` keywords. All classes and interfaces are reference types.

Value Types

Value types can be directly allocated on the stack and memory-wise consists only from the actual value. Furthermore, they can be either (i) built-in, for example primitive types like `int`, `long`, `float`, `bool` or (ii) user-defined, which stands for all custom structs. Some value types, for example many of the primitive types, are blittable, which means that they have the same managed and native representation. Additionally, value types are always passed by value – its content is copied, unless programmer uses the `ref`, `in` or `out` keyword. Similarly to reference types, these keywords can be only used in a few cases, for example when defining method parameters. It is also important to add that language designers made this feature in a way that it does not break memory safety of programs as described in Section 3.1.

3.4 Programming using CIL

In the following sections, we are going to describe basic programming using CIL. We start with describing metadata and then continue with several simple object-

oriented constructs. Later we advance to more complex examples, such as generics and usage of prefix instructions. This will help us in the following chapter when describing the design of instrumentation routines for dynamic analysis.

3.4.1 Metadata

In .NET, when programming using high-level languages, programmers do not usually need to fully understand how metadata work. Apart from the fact that users may specify custom attributes, metadata are generated by compilers.

The basic idea behind embedded metadata is to allow generation of a self-describing code. This means that programmers do not need to supply interface definition files. However, when programming in CIL, metadata become quite important. Due to this fact, in the following paragraphs, we describe the format of .NET programs – the Portable Executable (PE) format, we also discuss how are metadata stored, as well as introduce basic terminology common for .NET metadata records.

Portable Executable Format

Portable Executable (PE) [18] format is an executable file format used commonly on Windows operating system and for .NET executables. When inspecting .NET modules, we can observe that their PE sections contain apart from already mentioned CIL code also metadata tables.

Metadata tables

The purpose of metadata tables is to list all types and members defined by the current module. Moreover, it also lists all references to types and members defined by other modules that are referenced by the current module. Although there are many different kinds of metadata tokens, we focus only on the most common ones to give an example on what is stored in metadata tables.

- **`TypeDef`** stands for a type definition provided by the current module
- **`MethodDef`** stands for a method definition that is owned by a `TypeDef`
- **`FieldDef`** stands for a field definition that is owned by a `TypeDef`
- **`TypeRef`** represents an imported type that is owned by a different module
- **`MemberRef`** represents an imported method, field or a property that is owned by a different module.
- **`AssemblyRef`** stands for an imported external assembly

While this was certainly not a comprehensive listing of all metadata token types, these are the most important ones for basic programming in CIL. Later when discussing generics, we will introduce a few more that are specific when working with generic parameters, instantiated types and instantiated methods. But first, in the couple of the following subsections, we will show programming in CIL illustrated using basic object-oriented constructs.

3.4.2 Creating Objects and Structs

In case we want to create a new object that has a reference type, we should use the instruction `newobj`. This results in a memory allocation, zero-initialization of its fields and a call to the specified constructor. After these steps are finished, a reference to the new instance is pushed onto the evaluation stack. Usage of this instruction can be seen in the following code-listing.

```
IL_0000: newobj instance void [DeclaringAssembly]Type::.ctor()
```

With value types the situation is a bit more complex. Even though the `newobj` instruction can be also used with value types, in many cases this creates an unnecessary copy of the underlying value. Usually, we only need to store something to a local variable – memory that is available on the stack and is not accessible from different threads. In such cases we can perform a copy-elision optimization using instructions `initobj` and `call`.

```
// Example with initobj (zeroed value type)
IL_0000: ldloca.s 0
IL_0002: initobj valuetype [DeclaringAssembly]Type
```

```
// Example with call (initialized value type)
IL_0000: ldloca.s 0
IL_0002: call instance void valuetype [DeclaringAssembly]Type
    ::.ctor()
```

In both cases we first load an address of a local variable using the instruction `ldloca`. Then we either zero out the memory using the instruction `initiobj` or initialize it by calling the specified constructor with the `call` instruction.

Presented examples are the most common constructs when creating new objects and structs. However, there are some other ways, for example when creating arrays, which will be discussed in the following subsection.

3.4.3 Arrays

When dealing with arrays, there are several instructions programmers need to be aware of. For example, when creating arrays we cannot use the `newobj` instruction since arrays are considered a special case. Instead we need to use the instruction `newarr`. The instruction functions similarly to the `newobj` from the programmers point of view – it allocates memory for the array and zero-initializes its elements.

```
// Creating Int32 array of 100 elements and storing it to
    variable with index 0
IL_0000: ldc.i4 100
IL_0005: newarr [System.Private.CoreLib]System.Int32
IL_000a: stloc.0
```

Reading elements can be performed with instructions `ldelem.*` or `ldelema`, depending on whether we load the element by value or its address. An example of `ldelem.i4` instruction that loads an element of type `System.Int32` can be seen in the following code-listing.

```
// Loads previously stored array from variable with index 0
    and gets its 11-th element
IL_000b: ldloc.0
IL_000c: ldc.i4.s 11
IL_000d: ldelem.i4
```

Analogically, writing to array elements can be performed with instructions `stelem.*` as we show in the following code-listing.

```
1 // Loads previously stored array from variable with index 0
      and writes value 22 to its 11-th element
2 IL_000b: ldloc.0
3 IL_000c: ldc.i4.s 11
4 IL_000e: ldc.i4.s 22
5 IL_0010: stelem.i4
```

Using the code-listings above, we presented basic use cases in connection with arrays. Later, when discussing prefix instructions, we will briefly revisit arrays as there is one special case that will be important to look into. But now we will continue with our description of CIL.

3.4.4 Fields

There are two basic kinds of fields – static and instance fields. We will start by discussing static fields as they are not bound to a specific instance. An example of working with static fields can be seen in the following code-listing.

```
1 // Loads value of a field to local variable with index 0 and
      stores string "NewValue" back to the field
2 IL_0000: ldsfld string Class::StaticField
3 IL_0005: stloc.0
4 IL_0006: ldstr "NewValue"
5 IL_000b: stsfld string Class::StaticField
```

When working with instance fields, the situation is only a bit more complicated. In order to be able to work with an instance field, an instance needs to be specified in the form of pushing `this` onto the evaluation stack. We can accomplish it using the instruction `ldarg.0` because each instance method has 0-th argument, which is sometimes referred to as ”hidden this”. Usage of the `ldarg.0` instruction is common when retrieving the instance, using which the method was invoked. An example of how to work with instance fields can be observed in the following code-listing.

```
1 // Loads value of a field to local variable with index 0 and
      stores 11 back to the field
2 IL_0000: ldarg.0
3 IL_0001: ldfld int32 Class::InstanceField
4 IL_0006: stloc.0
5 IL_0007: ldarg.0
6 IL_0008: ldc.i4.s 11
7 IL_000a: stfld int32 Class::InstanceField
```

For completeness, there are two additional instructions that can be used to push address of a field onto the evaluation stack: `ldflda` and `ldsflda`. Similarly to arrays, we will briefly revisit fields once we get to prefix instructions which can change semantics and also the way we write and generate code. But first we need to look into more basic concepts, such as method calls and value type boxing.

3.4.5 Method Calls

We begin describing method calls by thoroughly discussing calling conventions and different call types. In order to invoke a method, we can use one of the

following instructions: `call`, `callvirt` and `calli`. Despite the fact that `calli`, indirect call, is at the time of writing this thesis not used by common compilers, there is still a lot to discuss about method calls using the other two call instructions.

Calling Convention

CIL provides a virtual calling convention, which is an uniform way for calling methods and as such it is completely independent from the underlying platform. In fact, it is the job of the JIT compiler to convert method calls to the actual calling convention, based on the given platform. Therefore, when programming in CIL, we always use the virtual calling convention whose rules are following [16]:

- All arguments are present on the evaluation stack
- When calling an instance method, the first argument is the `this` pointer
- Remaining arguments are pushed on the evaluation stack from left to right. This means the first real argument of the method is the lowest one on the stack, following the `this` pointer if available.

The resulting calling convention generated by the JIT compiler can be changed when directly calling native code. This can be achieved, for example using the `CallingConvention` field on the `DllImportAttribute`. The attribute forces the JIT compiler to generate calling convention for the given method exactly as specified by a programmer.

Instruction Call

The purpose of the `call` instruction is to invoke a method which is determined solely based on the provided metadata token. This means that the invoked method is not resolved based on the type of the receiver object on the stack if it is available. An example of a method call using the `call` instruction can be seen in the following code-listing.

```
1 // Example call to Console::WriteLine with multiple arguments
2 IL_0000: ldstr "Hello"
3 IL_0005: ldstr " "
4 IL_000a: ldstr "world!"
5 IL_000f: call void [System.Console]System.Console::WriteLine(
      string, object, object)
```

This instruction can be also used on delegates, or even for virtual methods, however, the same rule applies – the invoked method is determined at compile time. Whenever we want to use the actual type and its record in the virtual method table, we should use the instruction `callvirt` which will be discussed in the following paragraphs.

Instruction Callvirt

As already mentioned, the major difference between the `callvirt` and the `call` instructions is determining which method to invoke. With the `callvirt`

instruction the invoked method is resolved during runtime based on the receiver object. An example source code in C# where compilers must generate this instruction can be seen in the following code-listing.

```
1 class A { public virtual string M() { return "A"; } }
2
3 class B : A { public override string M() { return "B"; } }
4
5 class Example {
6   public void Method() {
7     var A = new B();
8     A.M();
9   }
10 }
```

The disassembled body of `Example::Method()` can be seen in the following code-listing. Note that even though we provided the metadata token for the method `A::M()`, during runtime the method `B:M()` will be called instead.

```
1 IL_0000: newobj instance void B::.ctor()
2 IL_0005: callvirt instance string A::M()
```

We showed basic usage for the instructions `call` and `callvirt` which at the time of writing this thesis covers all commonly used method invocation types. Previously we also mentioned the instruction `calli`, however, this instruction is currently not used by any available compilers and as such will not be further interesting for us. In the following paragraphs, we will focus on currently used constructs when calling native code.

Calling native code

While it is advised to stay in managed code as much as possible, there are certainly many cases when it is not possible. To mention a few examples, we can consider programming wrappers for native libraries or communicating directly with the CLR. Both cases are actually commonly used in certain framework assemblies that provide managed facades for low-level API, such as working with file system, networking or directly accessing some CLR features like GC.

There are multiple ways to call into native code, mainly QCalls, FCalls and P/Invokes. All of the mentioned approaches have one important thing in common – the target method is not implemented in CIL and programmers provide only method signature and some additional metadata in form of attributes. QCalls and FCalls are mechanisms to call the CLR. QCalls are identified as static extern methods to a library called QCall. FCalls are extern methods with `MethodImplOptionsAttribute` set to `InternalCall` and are more prone to cause errors in code. On the other hand, P/Invoke is a general mechanism for calling native libraries [17].

The mentioned approaches make it hard to further inspect or instrument implementation of certain methods as it would require to analyse also native code. When designing instrumentation routines, we need to keep in mind that calls to native code need to be handled carefully.

Conclusion

We discussed instructions for calling methods, virtual calling convention used in CIL, as well as presented some facts which might complicate the instrumentation process. Similarly to previous subsections, we will briefly revisit method calls when discussing prefix instructions. But first we will introduce two more important concepts – value type boxing and generics.

3.4.6 Boxing and Unboxing

Sometimes it is useful to be able to treat even value types like reference types. This can be helpful, for example, when passing value types as arguments to methods which expect `System.Object` or when calling methods defined by base classes, such as `ToString()` or `GetHashCode()`. Due to this reason. all value types are derived from class `System.ValueType` – this class is, however, not directly used when defining new value types. Instead language designers provided us with different ways to define them, for example, in C# we use the keyword `struct`.

For each value type, the CTS, which is described in Section 3.3.1, defines its corresponding boxed type which is a reference type. Whenever a value is boxed, an instance of its corresponding boxed type is populated with a bitwise copy of the original value [16]. This is always a result of executing the `box` instruction because boxed types cannot be directly referenced by their names.

Similarly for each boxed type, there is the `unbox` instruction which returns the boxed value as a value type. In this case it is not required to create a copy of the value, in fact it is common to return the address of the value type that is present in the boxed object [16].

3.4.7 Generics

This subsection is dedicated to discussing generics and showcasing how we can use them in CIL. We already introduced some metadata token types in Section 3.4.1, however, in order to generate code with instantiated types and methods, we need to look also into other metadata token types.

- **`GenericParam`** is a definition of a generic parameter owned by a `MethodDef` or a `TypeDef`
- **`GenericParamConstraint`** records constraints for a `GenericParam`. Each `GenericParam` can derive from a class or implement some interfaces
- **`TypeSpec`** is an instantiated type created from a generic `TypeDef`. It was created by substituting each `GenericParam` owned by the `TypeDef` with a concrete type
- **`MethodSpec`** is an instantiated method created from a generic `MethodDef`. It was created by substituting each `GenericParam` owned by the `MethodDef` with a concrete type

The key information from this subsection is the difference between the `TypeDef` and the `TypeSpec`, as well as between the `MethodDef` and the `MethodSpec`. In order to illustrate the difference, let us first consider the following code-listing in C#.

```
// Example: definition of GenericClass`2 and its instantiation
public class GenericClass<T, U> {
  public GenericClass(T item1, U item2) { }
}

public class Program {
  public void Main() {
    var item = new GenericClass<int, String>(11, "Hello");
  }
}
```

In the previous example, we can see the definition of a type `GenericClass`2`, which is a generic `TypeDef` that owns two `GenericParam`s. Its constructor is a `MethodDef` that owns no generic parameters. In the following code-listing we show the disassembled version of the constructor. Note the generic parameters which are present in the method signature.

```
// Disassembled constructor of the GenericClass`2
.method public hidebysig specialname rtspecialname instance
    void .ctor (!T item1, !U item2) cil managed {
  .maxstack 8

  IL_0000: ldarg.0
  IL_0001: call instance void [System.Private.CoreLib]System.
      Object::.ctor()
  IL_0006: ret
}
```

The main method of the first presented code-listing gets disassembled into the code which we show in the next code-listing.

```
// Disassembled Main method of the example
.method public hidebysig instance void Main () cil managed {
  .maxstack 8

  IL_0000: ldc.i4.s 11
  IL_0002: ldstr "Hello"
  IL_0007: newobj instance void class GenericClass`2<int32,
      string>::.ctor(!0, !1)
  IL_000c: pop
  IL_000d: ret
}
```

In the previous code-listing, we can observe that the `GenericClass`2` was already correctly instanced using types `System.Int32` and `System.String` and as such represents a `TypeSpec`. The constructor itself is not a generic method, since the mentioned generic arguments are owned by the `TypeDef` and not by the `MethodDef`. Therefore this method can is not a `MethodSpec`.

Whenever instrumenting code, SharpDetect needs to work with non-instantiated versions of types and methods because their CIL code is shared between all instantiations. For example, consider that we want to instrument a

method based on a method call that was invoked using a `MethodSpec`. In this case, we must first resolve the `MethodSpec` to its generic `MethodDef` and instrument the method's body there. Subsequently, when generating code that uses generics, they must be instantiated first. And now since we already covered the basic usage of generics, we can continue with the next important concept in CIL, which is prefix instructions.

3.4.8 Prefix instructions

Until now we were considering CIL instructions to be independent operations. However, it turns out that the basic behaviour of certain instructions is in some cases not ideal. While the most straight-forward solution would be to provide multiple variants of these instructions, this could quickly result in a significantly larger instruction set with many instructions being rarely used. Instead CIL introduces six special prefix instructions: `constrained`, `no`, `readonly`, `tail`, `unaligned` and `volatile`. These instructions can be used in well-defined situations to override the behaviour of some instructions when required.

Prefix instructions are not valid on their own and therefore cannot be considered as independent operations. In fact they must always be followed by certain instructions according to the specification [16]. Also when using control-flow branches, any prefix instruction is a valid label, but its subsequent instruction cannot be a label. In the following paragraphs, we will further describe prefix instructions and discuss how they modify the default behaviour of their subsequent instructions.

Instruction Constrained

The `constrained` instruction is used to call virtual methods on a type which was constrained to be type `T`. Therefore, this instruction must be followed by the instruction `callvirt` and works uniformly on reference types and value types.

Originally, the instruction `callvirt` could be applied only on reference types. However, when writing generic code and interfaced value types, it is sometimes necessary to actually use `callvirt` instruction also on value types. Moreover, `constrained` can be sometimes also used to avoid boxing when calling, for example methods defined by `System.Object` [16]. This can provide significant performance gain as there is no need to allocate new object on the heap and perform bitwise copy of the value type as described in Section 3.4.6. An example of code in C# that uses the `constrained` prefix instruction when compiled can be seen in the following code-listing.

```
1 // Example C# code that generates constrained call
2 struct CustomValueType { }
3
4 class Program {
5   static void Main() {
6     var instance = new CustomValueType();
7     instance.ToString();
8   }
9 }
```

In the code-listing above, we can see a definition of a value type called `CustomValueType` and an implementation of a method `Main()`. In the method

we create an instance of the `CustomValueType`, on which we then call the method `ToString()`. The disassembled version of this method can be seen in the following code-listing.

```
// Disassembled body of Program::Main() method:
IL_0000: ldloca.s 0
IL_0002: initobj CustomValueType
IL_0008: ldloca.s 0
IL_000a: constrained. CustomValueType
IL_0010: callvirt instance string [System.Private.CoreLib]
    System.Object::ToString()
IL_0015: pop
IL_0016: ret
```

In the previous code-listing, we can see the disassembled body of the method `Main()`, which shows usage of the `constrained` prefix instruction to call method `ToString()` on an instance of the `CustomValueType`. Even though boxing the instance would be a valid operation as described in Section 3.4.6, compilers prefer the variant using the `constrained` instruction as it results in a faster code.

Instruction Readonly

If we tried to access an address of an array element in a generic method, for example, from an array declared as `T[]` where `T` is a generic argument, we would run into runtime issues due to a type check defined for the `ldelema` instruction. This is something which can be avoided using the prefix instruction `readonly`. However, its main purpose is to push onto the evaluation stack a controlled-mutability managed pointer. Due to the fact that by using a regular managed pointer we are actually able to mutate the returned reference, the `readonly` prefix must be used to ensure that users cannot corrupt their memory [16].

This prefix instruction is valid only before the `ldelema` instruction and its usage can be seen in the following code-listing.

```
// Example code that generates readonly instruction
public class Program {
  public void Method<T>() {
    T[] array = new T[10];
    array[0].ToString();
  }
}
```

```
// Disassembled body of Program::Method()
IL_0000: ldc.i4.s 10
IL_0002: newarr !!T
IL_0007: ldc.i4.0
IL_0008: readonly.
IL_000a: ldelema !!T
IL_000f: constrained. !!T
IL_0015: callvirt instance string [System.Private.CoreLib]
    System.Object::ToString()
```

In the first part of the example presented above, we can see an implementation of a generic method `Program::Method<T>` written in C#. In the body of the method we created an array of generic type `T` and size 10. After that we called the method `ToString()` on its first element.

Using the second part of the example, we can observe the disassembled version of the generic method `Program::Method<T>`. We can see that when loading the address to the first element, compiler generated the `readonly` prefix instruction.

Instruction Tail

The ability to perform optimized tail calls is really important especially for functional programming languages, such as F#. Calling the same method before its return is a special case of recursion, called tail recursion. In order to support tail recursion, CIL introduces the prefix instruction `tail` that can be written before any of the call instructions. Without the prefix instruction `tail`, recursion could end with `StackOverflowException`.

The rules for using the `tail` instruction are fairly simple. Before the tail call, only arguments for the call can be pushed on the evaluation stack. Subsequently, right after the call, the only valid instruction is the `ret` instruction and the evaluation stack should be in the original state from the beginning of the current call. This is required because the way tail calls work in the CLR is by removing the current stack frame right before invoking the tail call [16].

While the `tail` prefix instruction is supported by the majority of implementations, compilers often do not generate this instruction at all. In fact at the time of writing this thesis, the most used C# compilers do not generate this instruction at all. Furthermore, F# compiler generates the instruction only in case the method can not be rewritten using a cycle, as can be seen in the following example [19]:

```
1   let apply f x = f x
```

In the previous code-listing, we can see a function that takes two arguments: a function and an argument. Its implementation then just simply calls the function using the provided argument. The disassembled version of the function can be seen in the following code-listing.

```
1 IL_0000: ldarg.0
2 IL_0001: ldarg.1
3 IL_0002: tail.
4 IL_0004: callvirt instance !1 class [FSharp.Core]Microsoft.
     FSharp.Core.FSharpFunc`2<!!a, !!b>::Invoke(!0)
5 IL_0009: ret
```

From the disassembled code above, we can see that the compiler actually generated a true tail call using the `tail` prefix instruction. All arguments are present on the evaluation stack before the tail call is invoked and immediately after it returns, there is only the `ret` instruction.

Instruction Unaligned

Whenever we have a pointer on the evaluation stack that is not aligned to the natural pointer size, to access this data we need to use the `unaligned` prefix instruction [16]. This can be used before instructions, such as field reads and writes or some special instructions like the `cpblk`.

Instruction Volatile

In order to perform a volatile read or write, we need to prefix the memory operation with the `volatile` prefix instruction. Additionally, the `volatile` instruction can be combined with the `unaligned` instruction – the only exception are static fields which can have only the `volatile` prefix [16]. An example can be seen in the following code-listing.

```
// Declaration of an instance volatile field
.field private int32 modreq([System.Private.CoreLib]System.
    Runtime.CompilerServices.IsVolatile) Field
```

```
// Disassembled method reading the volatile field
IL_0000: ldarg.0
IL_0001: volatile.
IL_0003: ldfld int32 modreq([System.Private.CoreLib]System.
    Runtime.CompilerServices.IsVolatile) Class::Field
```

In the previous code-listing, we can see an example of using the `volatile` prefix instruction before the `ldfld` instruction making the access to the field volatile. Moreover, we can notice an important custom modifier on the field declaration which also states that the field is volatile:

```
modreq([System.Private.CoreLib] System.Runtime.
    CompilerServices.IsVolatile)
```

The custom modifier is embedded into the field declaration as metadata and therefore provides information for other modules on how to treat this field. We can say that the prefix instruction `volatile` is information for the CLR, while the custom modifier is mostly used by compilers.

Instruction No.*check

The `no` instruction can be used in one of its following three forms: `no.typecheck`, `no.rangecheck` and `no.nullcheck`. These instructions can be used to instruct the CLR that it may skip some runtime checks that are normally performed during execution. The first form of the instruction indicates that type checking can be skipped and can be used before instructions such as `castclass` or `unbox`. The second form of the instruction indicates that range checking can be skipped and can be used before array instructions that manipulate with array elements. Finally, the third form of the instruction indicates that null checks can be skipped and can be used, for example when loading fields, calling virtual methods or getting addresses of array elements [16].

However, usage of these instructions is generally not considered safe and always results in an unverifiable code as described in Section 3.1. While these instructions can provide some performance improvements, the CLR is able to ignore them and still throw exceptions if a specific check fails [16].

Summary

In the previous paragraphs, we described available prefix instruction, how they change the behaviour of individual instructions and how well they are supported by current CLR implementations. Knowledge of prefix instructions will be crucial when designing instrumentation routines for dynamic analysis, as there are many

constraints on how the resulting code has to look like when it comes to using these constructs.

3.4.9 Flow Control

This subsection describes instructions that can be used to introduce decision making, loops and branches when programming using CIL. Similarly to other programming languages, these constructs are used for conditional execution of blocks of code.

In order to create a conditional branch, the following instructions can be used: `beq`, `bge`, `bgt`, `ble`, `blt`, `bne`, `brfalse`, `brinst`, `brnull`, `brtrue` and `brzero`. All of the mentioned instructions also exist in a short-form where the jump target can be specified using a single byte. In the case of the default form of instructions, the jump target is specified using four bytes. Furthermore, branching instructions that do not branch based on an equality test also exist in a special form that is used for unsigned and floating types [16]. A simple branching example can be seen in the following code-listing.

```
1  // if (arg1 <= arg2)
2     IL_0000: ldarg.1
3     IL_0001: ldarg.2
4     IL_0002: bgt.s IL_0006:
5     {
6       // return arg1;
7       IL_0004: ldarg.1
8       IL_0005: ret
9     }
10 // else
11    {
12        // return arg2;
13      IL_0006: ldarg.2
14      IL_0007: ret
15    }
```

In the previous code-listing, we can see an implementation of a simple compare method that takes two signed integers as its input and on its return yields the smaller of the two values. Note that this is implemented using the instruction `bgt.s`, which stands for *branch greater than* and also it is the short-form of the instruction because its target is just two instructions apart. Therefore, operands of the comparison are determined based on their position on the evaluation stack – in our case the first operand is `arg2` and the second is `arg1`.

CIL supports also unconditional branches using the following two instructions: `br` and `jmp`. The difference between these instructions is, however, really significant. While the `br` instruction is quite similar to the previous instructions (except for the missing condition), the `jmp` instruction takes as its operand a method metadata token. When the `jmp` instruction is executed, the evaluation stack must be empty and the calling convention and arguments of the target method must match the current method. Similarly to the `tail` instruction described in Section 3.4.8, the current stack frame is removed after the execution of the `jmp` instruction. Furthermore, usage of the `jmp` instruction always results in an unverifiable code [16].

There is a common restriction when using branching instructions. These

instructions can not be used to jump out of certain code regions, such as synchronized blocks or handler blocks.

3.4.10 Exceptions

Since we already covered flow control in the previous subsection, the important concepts that remain to be explained are handler blocks and exceptions. In CIL, the exceptions are quite straight-forward – in order to throw an exception, the following two instructions can be used: `throw` and `rethrow`. However, the instruction `rethrow` can be only used to throw an already caught exception within a catch block. Therefore, in order to throw a new exception, the instruction `throw` needs to be used. An example of throwing a new exception of type `System.InvalidOperationException` can be seen in the following code-listing.

```
1 IL_0000: ldstr "The requested operation is not valid"
2 IL_0005: newobj instance void [System.Private.CoreLib]System.
     InvalidOperationException::.ctor(string)
3 IL_000a: throw
```

While technically the CLI permits any object to be thrown using these instructions, according to the specification [16] only classes derived from `System.Exception` are permitted in this context. Now that we covered the basic usage of exceptions, we continue by describing handler blocks.

3.4.11 Handler Blocks

Any method can have some of its instructions marked as *protected*, which stands for a `try` block in high-level programming languages. Moreover, each `try` block can be associated with one or multiple handler blocks [16]. There are four types of handler blocks: `filter` blocks, `catch` blocks, `finally` blocks and `fault` blocks, all of which will be described in the following lines.

In order to mark a certain block of instructions as either a `try` block, or one of the handler blocks, the beginning of the block should be the first affected instruction and the ending of the block should be the first instruction after the block. This results in a system where no two handler blocks can have the same starting address [16]. An example of a simple try-catch construct can be seen in the following code-listing.

```
1 .try {
2   IL_0000: ldstr "The requested operation is not valid"
3   IL_0005: newobj instance void [System.Private.CoreLib]System
       .InvalidOperationException::.ctor(string)
4   IL_000a: throw
5 } // endtry (IL_0000 - IL_000b)
6
7 catch [System.Private.CoreLib]System.Exception {
8   IL_000b: pop
9   IL_000c: leave.s IL_0018:
10 } // endcatch (IL_000b - IL_000e)
11
12 IL_000e: ret
```

In the presented code-listing above, we can see the same code as in the previous code-listing, but marked as protected code, which forms a `try` block. Its associated `catch` block does nothing, but it is important to note that it must be exited using the instruction `leave` or its short-form `leave.s`.

In the previous example, we presented a `catch` block handler that catches managed exceptions compliant with the specification [16]. If we wanted to further constrain when should the `catch` block handler be executed, we could additionally specify also a `filter` block handler. An example of setting a filtering constraint that checks whether the `Message` property of the exception is not `null` can be seen in the following code-listing.

```
filter {
  IL_000b: isinst [System.Private.CoreLib]System.Exception
  IL_0010: dup
  IL_0011: brtrue.s IL_0017:
  IL_0013: pop
  IL_0014: ldc.i4.0
  IL_0015: br.s IL_0022:
  IL_0017: callvirt instance string [System.Private.CoreLib]
     System.Exception::get_Message()
  IL_001c: ldnull
  IL_001d: cgt.un
  IL_001f: ldc.i4.0
  IL_0020: cgt.un
  IL_0022: endfilter
} // endfilter (IL_000b - IL_0024)
```

In the code-listing above, we can see that the `filter` block is ending with the instruction `endfilter`. Moreover, the `endfilter` instruction pops a single item of type `System.Int32` from the evaluation stack, its value must be either `1` or `0`. If the value was equal to `1`, the execution continues with the `catch` block. Otherwise the execution searches for a different `catch` block.

The next handler block, the `finally` block, is executed always, regardless of whether an exception occurred or not. Any exception handling, as presented within the previous example, must be nested in a `try` block, otherwise the `finally` block might not be executed correctly. Furthermore, it would result in an unverifiable code [20]. An example can be seen in the following code-listing.

```
.try {
  .try {
    /* protected instructions */
  }
  catch [System.Private.CoreLib]System.Exception {
    /* exception handling */
  }
  leave exitProtected
}
finally {
  /* finally handling */
  endfinally
}
exitProtected:
```

The last type of handler block, the `fault` block, is quite similar to `finally`. However, it is executed only if an exception occurred within the original `try` block. It can be written using the `fault` directive and always ends with the instruction

`endfault`. Furthermore, there is actually no way to express this handler block using C#.

3.5 Compilation to Native Code

As we already mentioned, CIL code cannot be directly executed by the CLR and therefore it needs to be compiled into native code prior to the execution. There are two options how the CLR can obtain native code, first being native images and second being JIT compilation.

3.5.1 Native Images

Native images, also known as ahead of time (AOT) compilation, contain executable native code which is similar to what would the JIT compiler produce under normal circumstances. This approach can be used to minimize start-up time of applications, which is commonly used for some framework assemblies.

For .NET Core we can generate native images using a tool called CrossGen [21]. Whenever there is a native image available, the CLR prefers the load of that specific image. Furthermore, when a native image is loaded by the CLR, its code can be executed without any interception of the JIT compiler.

Despite the mentioned advantages, there are also some drawbacks when working with native images. For example, size – native images are significantly bigger than managed assemblies. Moreover, native images are dependent on the platform that they were generated for.

3.5.2 JIT compiler

The preferred alternative to native images is JIT compilation. Whenever the CLR tries to load an assembly for which it was unable to find its native image, it defaults to JIT compilation of the assembly.

When loading managed assemblies, method tables of types, specifically their individual entries are initialized with a call to the JIT compiler. Therefore, whenever we call a method for the first time, it is handled by the JIT compiler. It is responsible for generating native code and for patching the entry in the method table of its declaring type. The patch is really simple – it needs to ensures that consecutive calls jump straight into the generated native code [22].

The most important observations are that methods are compiled just before they are executed and that each method is compiled at most once. This is different compared to the process of native image generation where we needed to compile all referenced methods, because we could not predict what is needed at runtime and what can be skipped.

3.5.3 Optimizations

Both mentioned approaches can apply various optimizations on the resulting native code. These optimizations are not as powerful as, for example optimizations issued by moderns C++ compilers. Nonetheless, in many cases they are capable of changing code in a significant way in order to make it faster.

3.6 Managed Code Hosting

Now that we covered the most important components of the CLI, we can discuss how we can execute .NET programs. Since we already mentioned in Section 3.3 that code written in CIL cannot be directly executed, we are going to need a hosting application.

The purpose of the hosting application is to load and initialize the CLR and then start the execution of the provided assembly. Luckily, most programmers will never need to write their own hosts because there is already a viable implementation called `dotnet` [23]. This host is capable of many more operations than just running .NET assemblies, to name a few: building, running tests, packing nugets or publishing self-contained packages.

Even though the initialization process of the CLR happens mostly in native code, there is one particularly interesting assembly that plays an important role during the initialization – the core library. This library, more specifically `System.Private.CoreLib` will be further discussed in the following section. After that we will revisit the `dotnet` command, especially publishing self-contained packages where we will showcase how we can access all managed and native dependencies of any .NET Core program.

3.7 System.Private.CoreLib

Core libraries, such as `System.Private.CoreLib` for .NET Core, have several unique properties compared to other managed assemblies. This is mostly due to the fact that core libraries are always tightly coupled with the CLR and implement the most basic types and methods provided by the base class library. The most important constraints and properties of core libraries are the following [17]:

- Base types, such as, `System.Object`, `System.Int32` or `System.IntPtr` are always defined in core libraries. These types are also needed by the CLR and therefore core libraries are always loaded during the startup phase of the CLR.
- Core libraries cannot have any managed dependencies and they may use only native libraries. Additionally, only one core library can be loaded per process.
- Core libraries need to expose certain functions from the CLR and are heavily used for native interoperability. Among others, this requires that managed types can be easily mapped to their corresponding native types.

It is common to find core libraries installed in system together with corresponding native images, as described in Section 3.5.1. Being the common dependency for .NET programs, native images can provide significant performance gain during startup.

3.8 Self-contained Packages

Command `dotnet publish` is capable of packing applications together with all of their dependencies. As a result, it creates so called "self-contained packages" with all dependencies in a single folder. This is a user-friendly way to deploy .NET Core applications because it does not place any requirements on the target machine about installed frameworks or runtimes.

Packaging applications can also be a great utility when implementing instrumentation routines because one of the hardest challenges would be to find all referenced assemblies. Assemblies can be generally in many directories, not even counting different versions or platform-dependent assemblies. Therefore, having an utility that correctly resolves all references and places them in a single folder can be a great advantage.

Despite being able to access all dependencies, we might still run into issues when modifying assemblies on certain .NET implementations. The most commonly used mechanism which could prevents this is called strong-name verification and will be further discussed in the following section.

3.9 Strong-named Assemblies

Instrumentation routines for dynamic analysis will heavily rely on the ability to successfully load instrumented assemblies and execute them. In this section, we will describe how we can achieve this and why is this possible.

.NET Framework introduced strong assembly names which consist of information like filenames, versions and public keys. Their purpose was to tackle two kinds of problems – manageability and security. Strong assembly names provide unique identifiers and therefore counter issues like DLL hell [24]. Regarding security, strong assembly names can be used, for example to ensure that assemblies were not modified since build.

Despite the fact that using strong assembly names the CLR could check integrity of assemblies, in practise it was commonly bypassed – one of the reasons was performance. In fact since .NET Framework 3.5, the check was turned off by default when loading assemblies from trusted sources [25]. Newer implementations, like .NET Core, do not validate assemblies integrity during load at all. In order to ensure that modified assembly gets loaded by the CLR, we need to ensure that it comes up first when looking for the specific assembly based on its filename and version.

4. Design and Implementation of SharpDetect

SharpDetect is a framework for dynamic analysis of .NET Core programs. It takes executable assemblies as its input and yields instrumented assemblies together with environment configuration for dynamic analysis. In this chapter, we describe the design and implementation process of SharpDetect. Furthermore, we describe several interesting problems that we encountered during our work on this project and at the end of the chapter, we propose some improvements that can be implemented as future work.

4.1 Overview

SharpDetect consists of five modules, `SharpDetect.{Common, Console, Core, Injector, Plugins}`. Moreover, we can divide them into two major categories: compile-time and runtime modules. The first category consists of `Console` and `Injector`, while the runtime category includes `Core` and `Plugins`. Basically, the only exception from this rule is `Commmon`, since it is used both during compile-time and runtime. The basic overview of SharpDetect can be seen in Figure 4.1.

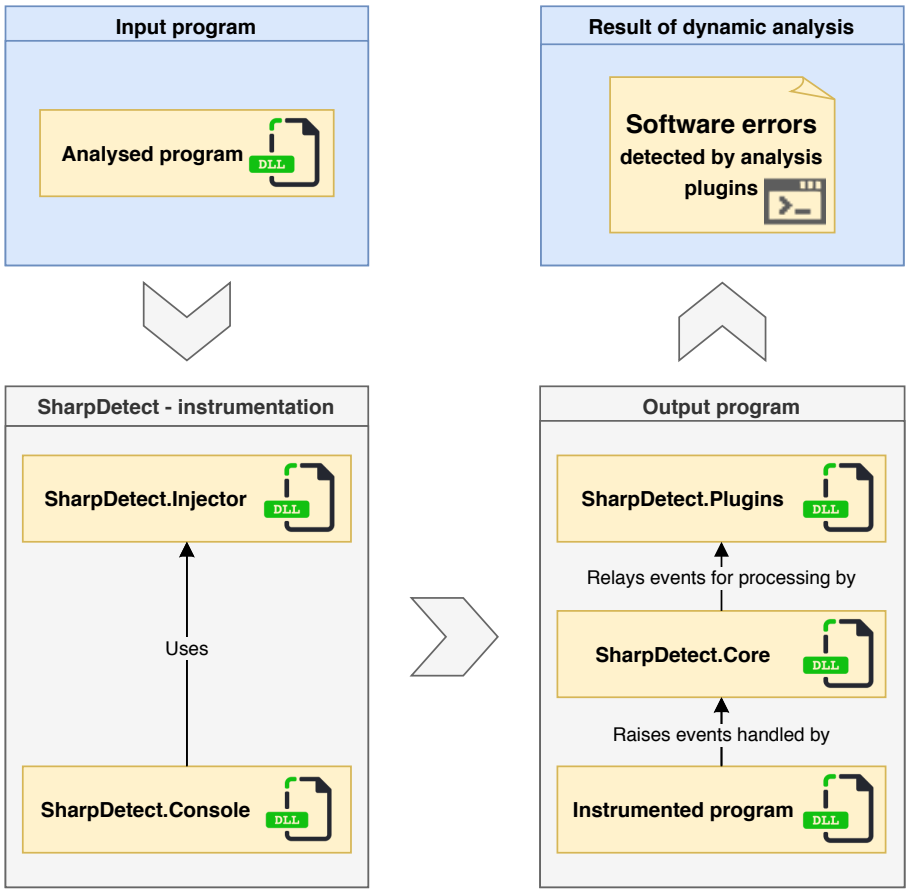

Figure 4.1: High-level overview of SharpDetect's architecture. There are two main parts: compile-time instrumentation and dynamic analysis execution.

4.1.1 SharpDetect.Console

`SharpDetect.Console` is a .NET Core frontend of SharpDetect that is implemented as a console application. Its main purpose is to create self-contained packages, as described in Section 3.8, from analysed .NET programs, run the instrumentation process and also execute the dynamic analysis using the instrumented assemblies.

An integral part of this module is parsing configuration from provided files and the command-line. When passing configuration to SharpDetect, there are actually multiple ways to do so. The first place where SharpDetect looks for configuration is the global configuration file `global-config.json` that has to be placed right next to the `SharpDetect.Console.dll`. The default content of this file can be seen in code-listing 4.1.

```
{
  "AlwaysIncludeMethodPatterns": [
    "System.Threading.Monitor",
    "System.Threading.Thread::.ctor",
    "System.Threading.Thread::Start",
    "System.Threading.Thread::Join"
  ],

  "ArrayInjectors": true,
  "FieldInjectors": true,
  "MethodInjectors": true,
  "ClassCreateInjector": true,
  "ObjectCreateInjector": true,

  "VerifyInstrumentation": true
}
```

Listing 4.1: Default content of the SharpDetect's global configuration file. All supported event types are instrumented and created assemblies are verified after the instrumentation. Furthermore, method patterns that are defined to be always included are always instrumented regardless of any overrides by other configurations.

Apart from the patterns that are always instrumented, the default configuration provided by the `global-config.json` can be easily overriden and extended by providing a local configuration that describes a concrete dynamic analysis of a specific program. Additionally, the local concrete configuration needs to specify the target assembly and patterns that should be instrumented from the specific assembly or one of its referenced assemblies. An example of such configuration can be seen in code-listing 4.2.

Correct usage of the global and local configuration files significantly reduces the amount of analysed events, and as such fulfils the goal **(G2)** about configurability described in Section 1.1. Moreover, we need to specify a path where SharpDetect is supposed to search for analysis plugins. This can be easily achieved by setting the environment variable `SHARPDETECT_PLUGINS` to a directory with plugins. This causes SharpDetect to search the specified directory, as well as its subdirectories, for available plugins.

Once the necessary configuration is prepared, `SharpDetect.Console` can be

used as can be seen in code-listing 4.3. To perform individual actions, `SharpDetect.Console` relies mostly on other modules that are described in the following subsections.

```
{
  "TargetAssembly": "Assembly.dll",

  "FieldPatterns": [
    "NamespaceA.TypeA",
  ],

  "MethodPatterns": [
    "NamespaceA.TypeA", "NamespaceB.TypeB::Method"
  ],

  "ArrayInjectors": false,
  "FieldInjectors": false,
}
```

Listing 4.2: Example content of a concrete local configuration for a specific dynamic analysis. Users have to specify a path to the target assembly and instrumentation patterns for methods and fields. On top of that users can override the default configuration provided by the `global-config.json`.

An important observation is that SharpDetect must actually traverse the whole call graph, regardless of the provided `MethodPattterns` and `FieldPatterns`. This is necessary in order for certain settings, such as `AlwaysIncludePatterns`, to work because many methods subjected to analysis might be nested inside a call graph segment that would be otherwise skipped based on other patterns.

```
// [Optional] prepare C# project for instrumentation
dotnet SharpDetect.Console.dll build <path_to_csproj> \
        --rid <platform_rid> --output <output_folder>

// Instrument assembly based on the provided configuration
dotnet SharpDetect.Console.dll instrument <path_to_config>

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

Listing 4.3: Example usage of `SharpDetect.Console` to build project, instrument the target assembly and run the dynamic analysis.

In order to run the dynamic analysis, users must also specify a plugin registration string. The format of the registration string is quite simple, it consists of one or multiple plugin names that are delimited with the `'|'` character. An example of the plugin registration string would be the following: `"plugin1|plugin2|...|pluginN"`. Registering plugins into chains is further described in the following subsection that is solely dedicated to plugins.

4.1.2 SharpDetect.Plugins

As we previously mentioned, SharpDetect also contains a few plugins, more specifically Eraser [7] and FastTrack [8], that can be used to detect and predict certain concurrency issues. Their implementation can be found within the module `SharpDetect.Plugins`. Additionally, it is loosely-coupled with the tool. In order to implement custom plugins, users must only ensure that their plugin implementation is derived from the abstract class `BasePlugin` defined by the module `SharpDetect.Core`.

Custom plugins can benefit from the fact that analysis events are supplied through virtual methods on the abstract `BasePlugin` class. In code-listing 4.4, we present a couple of these methods, which are available for overriding by custom plugins. For completeness, SharpDetect offers the following analysis events: analysis start, analysis end, array created, array element read, array element written, class constructed, field read, field written, lock acquire attempted, lock acquire returned, lock released, object created, object wait attempted, object wait returned, object pulsed one, object pulsed all, method called, method returned, user thread created, user thread started and user thread joined.

```
1  // Called right after SharpDetect initialization
2  public virtual void AnalysisStart(MethodDescriptor method);
3  // Called right before program exits from its entrypoint
4  public virtual void AnalysisEnd(MethodDescriptor method);
5
6  // Called when an array element is read
7  public virtual void ArrayElementRead(int threadId,
       MethodDescriptor method, Array array, int index);
8  // Called when an array element is updated
9  public virtual void ArrayElementWritten(int threadId,
       MethodDescriptor method, Array array, int index, object
       value);
10
11 // Called right before the execution of the return instruction
        of a class constructor - .cctor()
12 public virtual void ClassConstructed(int threadId,
       TypeDescriptor type);
13
14 // Called right before a lock method is called
15 public virtual void LockAcquireAttempted(int threadId,
       MethodDescriptor method, object lockObj, (int, object)[]
       parameters);
16 // Called right after a lock method returned
17 public virtual void LockAcquireReturned(int threadId,
       MethodDescriptor method, object lockObj, bool result, (int,
        object)[] parameters);
18
19 // Called right after the execution of the newobj instruction
       on reference type
20 public virtual void ObjectCreated(int threadId, object obj);
```

Listing 4.4: Example of some methods that can be found in the abstract class `BasePlugin` that can be overriden to implement custom analysis plugins.

The default implementation of all public virtual methods on the abstract class `BasePlugin` forwards the information about events to the next plugin in the chain

if such plugin is available. We already mentioned plugin chains in the previous subsection, more specifically how we can use them to hook multiple plugins for the dynamic analysis. Users are required to provide plugins registration string in the format `"plugin1|plugin2|...|pluginN"` that consists of names of required plugins delimitted by the `'|'` character. This basically means that whenever SharpDetect receives an analysis event, it is dispatched first to the `plugin1`. A plugin that got an event may choose to consume the event or forward the event to the next plugin in the chain after processing. Therefore, this is another way to filter analysis events that need to be processed. Nevertheless, the presented ability to implement custom analysis plugins fulfils the goal **(G1)** related to extensibility as described in Section 1.1.

4.1.3 SharpDetect.Core

The main component that is used at runtime is called `SharpDetect.Core`. During the initialization phase of the dynamic analysis, the purpose of this component is to register event handlers for the instrumented events and setup required plugins. After the initialization process is done, its main purpose is to orchestrate the dispatching of analysis events to plugins.

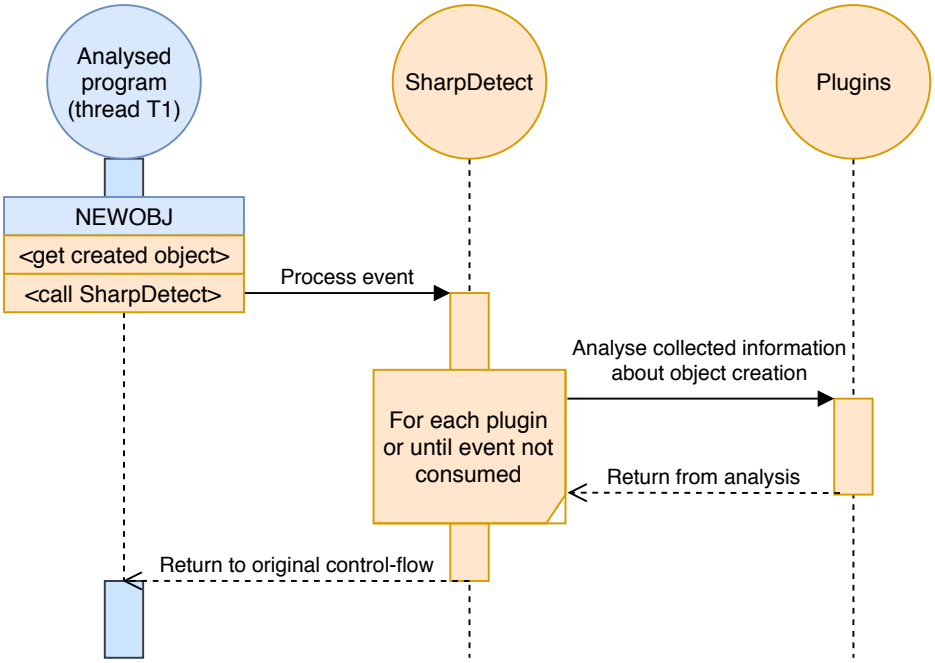

Figure 4.2: Basic principle of capturing and analysing events that is illustrated on the case of a dynamic object allocation. Right after an object is created, its reference is stored and passed to SharpDetect. SharpDetect then packs additional information, such as managed thread id of the thread that raised the event, and dispatches its information to the registered plugin chain. Blue-coloured items represent the content of uninstrumented assembly. Everything in the orange colour was added during the instrumentation process.

In order to initialize this module, `SharpDetect.Injector` must inject bootstrapping instructions at the beginning of analysed program's entrypoint

method. As previously mentioned, the purpose of these instructions is to call the initialization method of SharpDetect that is responsible for registering event handlers and preparing analysis plugins. The initialization method is basically a static instance getter on the singleton class `EventDispatcherFrontend`. Therefore, the bootstrapping code that needs to be instrumented at the beginning of the entrypoint method can be written in C# exactly as can be seen in the following code-listing.

```
1 SharpDetect.Core.EventDispatcherFrontend.GetInstance();
```

The presented code above can be easily translated into CIL using two instructions, `call` and `pop`, which can be seen in the following code-listing. We just need to make sure that SharpDetect assemblies are added to the referenced assemblies of the analysed program.

```
1 IL_0000: call class [SharpDetect.Core]SharpDetect.Core.
      EventDispatcherFrontend [SharpDetect.Core]SharpDetect.Core.
      EventDispatcherFrontend::GetInstance()
2 IL_0005: pop
```

Right after the initialization phase is completed, analysis events are captured and further dispatched to registered plugins. The basic principle how this works is illustrated in Figure 4.2 on an example where we instrumented dynamic object allocation, i.e., usage of the instruction `newobj` together with reference types.

From the figure above, we can see that analysis of individual events is always carried out by the same thread that raised them. Therefore, whenever a thread raises an event, its control-flow is hijacked to perform analysis of the event. Once the analysis is completed, the control-flow is returned to the analysed program. More information on how are events captured is presented in the following subsection.

4.1.4 SharpDetect.Injector

`SharpDetect.Injector` is responsible for instrumenting target assemblies based on the configuration obtained from `SharpDetect.Console`. After it instrumented target assemblies, it is no longer needed during runtime.

This module contains individual instrumentation routines for injecting custom classes, methods, fields and events, but also for modifying various existing methods in order to capture information about analysis events, as described in Section 2.1. The instrumentation process supports even framework assemblies, and overall it prepares necessary environment for the subsequent dynamic analysis.

In order to describe the instrumentation process to a greater detail, let us consider a method that allocates an object, which we intend to capture, as previously illustrated in Figure 4.2. Since we already covered object allocation using CIL in Section 3.4.2, we know that in order to create an object, we need to push constructor arguments before the `newobj` instruction. Then after executing the `newobj` instruction, a reference to the new instance is pushed onto the evaluation stack. First consider the following code-listing with a single instruction that creates an instance of `System.Object`.

```
IL_0000: newobj instance void [System.Private.CoreLib]System.
    Object::.ctor()
```

In order to notify `SharpDetect.Core` about the occurrence of this specific event, `SharpDetect.Injector` instruments code after the `newobj` instructions exactly as can be seen in code-listing 4.5. Presented code is created as a result of applying the class `SharpDetect.Injector.Injectors.ObjectCreateInjector` on the instruction shown in the code-listing above.

```
IL_0000: newobj instance void [System.Private.CoreLib]System.
    Object::.ctor()
// Duplicate reference to the newly created object instance
IL_0005: dup
// Store the top instance in the variable V_0
IL_0006: stloc.0
// Get instance to EventDispatcher backend
IL_0007: call class [System.Private.CoreLib]SharpDetect.
    EventDispatcher [System.Private.CoreLib]SharpDetect.
    EventDispatcher::get_Instance()
// Load stored object instance in the varriable V_0
IL_000C: ldloc.0
// Notify SharpDetect about object creation
IL_000D: call instance void [System.Private.CoreLib]
    SharpDetect.EventDispatcher::RaiseObjectCreate(object)
```

Listing 4.5: Example of instrumenting dynamic object allocation. SharpDetect duplicates the reference to the object instance on the evaluation stack and stores one copy to a local variable. Then a reference to the `EventDispatcher` together with the copied object reference gets pushed onto the evaluation stack and finally the ObjectCreate analysis event is raised.

When instrumenting CIL code, it is necessary to make sure (i) that the resulting code is valid, (ii) that after execution the code leaves the evaluation stack in its original state and (iii) that the `.maxstack` annotation of the method is adjusted if necessary. In order to set the annotation correctly, SharpDetect needs to be aware of how much stack the CLR needs when executing the instrumented code. If we look at the original uninstrumented code from the last example, it pushes exactly one item on the stack – a reference to the new object instance. Therefore, the original code reaches maximum stack depth of one. However, the instrumented code reaches at one point the depth of three items, which is why SharpDetect must increase the maximum stack depth. Furthermore, it needs to fix also the affected control flow branch instructions, as well as handler blocks.

Correcting Broken Branch Targets

Since we already covered control flow instructions in Section 3.4.9, it should be clear that by instrumenting code, we can break them quite easily. Consider a case where we have a brach target instruction, before which we need to inject some instructions. Now if the original instruction was a target for a branch instruction, our injected code would not be executed together with the original target instruction. This issue is shown in the following code-listing.

```
IL_0000: <injected_instruction>
IL_0001: <target_instruction>
// more code
IL_xxxx: br IL_0001:
```

As this is certainly not an intended behaviour, we need to fix branch targets whenever a situation like this one occurs. We need to make sure that whenever we inject code, it is always executed together with the instruction it was intended for. Otherwise, we could encounter issues like `System.InvalidProgramException` during the execution of the instrumented assembly.

Correcting Malformed Handler Blocks

Moreover, by injecting code SharpDetect can also break handler blocks that were described in Section 3.4.11. In the mentioned section, we described that each handler block must start with the first affected instruction and end with the first non-affected instruction. The cases that need to be fixed are the following:

- Injecting before an instruction that is at the beginning of a `try` block. In this case we want our code to become the new beginning of the `try` block. This is due to the fact that the original code was for some reason protected and therefore the injected code should be protected as well.
- Injecting before an instruction that is at the beginning of a handler block. This is a similar case to the previous one. We want our code to end up in the beginning of the specific handler block. Note that in this case we additionally need to fix also the ending of the previous block, which is either a `try` block or a handler block. The previous block needs to point at the beginning of the injected code.
- Injecting before an instruction that is the next instruction after the last handler block. In this case we want to exclude the code from the handler block. Therefore, SharpDetect needs to fix the ending of the handler block to point at the beginning of the injected code.

Furthermore, also occurrences of the `leave` instruction must be fixed according to the same rules because their target instruction changes exactly in the same way. We also need to make sure that we do not break prefix instructions, which were described in Section 3.4.8. The described corrections of handlers blocks ensure that whenever SharpDetect instruments code within handler blocks, the resulting code is valid.

4.1.5 SharpDetect.Common

The aim of the `SharpDetect.Common` module is to provide shared definitions, description of injected types and analysis events, as well as some utility classes and extension methods. This library is shared by most other modules and as such is needed both during compile-time and also when executing the dynamic analysis.

The definition of all analysis events can be located within the class `SharpDetect.Common.Definitions`. Basically, analysis events are public static

properties of the type `SharpDetect.Common.Events.DynamicAnalysisEvent` that have to be annotated with the `RegisteredEvent` attribute. Moreover, if it is an extension to an already existing analysis event, it needs to be annotated also with the `Extension` attribute, both from the namespace `SharpDetect.Common.Metadata`. An example definition of a few events can be seen in code-listing 4.6.

```
1 [RegisteredEvent(Type=AnalysisEventType.ObjectCreate,
      Parameters = new string[] { "System.Object" })]
2 public static AnalysisEvent ObjectCreate { get; set; }
3
4 [RegisteredEvent(Type=AnalysisEventType.MethodCall,
      Parameters = new string[] { "#PARAMS#", "System.String" })]
5 public static AnalysisEvent MethodCall { get; set; }
6
7 [Extension(FromEvent = DynamicAnalysisEventType.MethodCall)]
8 [RegisteredEvent(Type=AnalysisEventType.LockAcquireAttempt,
      Parameters = new string[] { "#PARAMS#", "System.String" })]
9 public static AnalysisEvent LockAcquireCall { get; set; }
```

Listing 4.6: Example definition of three analysis events: ObjectCreate, MethodCall and LockAcquireCall, which is an extension of MethodCall.

In the code-listing above, we can see that in order to define an analysis event, one needs to provide its name, event type and parameters. Moreover, when specifying parameters, users have to specify valid reflection names of type definitions as described in Section 3.4.1 about metadata. There is one exception – `#PARAMS#` – that stands for a type specialization rather than a type definition. It has the type `System.ValueTuple<System.Int32,System.Object>[]`, which is quite helpful when capturing method arguments. Furthermore, due to the reason that will be covered in the development diary (Section 4.3.5) when talking about modifying core libraries, the reflection names can refer only to types defined by the core library, i.e., `System.Private.CoreLib`.

Once the event is defined, a handler must be dynamically registered in the class `SharpDetect.Core.EventDispatcherFrontend`, and its corresponding injector must be implemented when it is a base analysis event. In case the added event is just an extension to a previously defined base analysis event, its descriptor should be added to `SharpDetect.Common.Extensions`. After all these steps are done, the new analysis event is ready to use.

4.2 Testing SharpDetect

In the previous subsection, we focused on the individual modules of SharpDetect that users of this tool interact with when performing dynamic analysis. However, for the development purposes there are actually two additional modules that are used for testing. Moreover, during our work on this project we defined a continuous integration pipeline using the Gitlab API [26] that further helped maintain the functional integrity of the project throughout the development. In the following subsections, we describe these modules and capabilities of the pipeline, as well as mention some tools that helped us during the development but are not directly integrated with SharpDetect.

4.2.1 Unit Tests and Functional Tests

SharpDetect contains two testing modules, `SharpDetect.UnitTests` and `SharpDetect.FunctionalTests`. The first module, as it name suggests, is dedicated to unit tests. This is used mostly for testing some analysis utilities from `SharpDetect.Core` and implementation of analysis plugins.

Unfortunately, code instrumentation is not that easy to test because instrumentation routines often access metadata tables when searching for various types and members. Also the default behaviour of .NET implementations is to load assemblies from a persistent storage, rather than directly from a memory stream. Due to this reasons, we developed a simple testing framework using the library NUnit [27] that, based on a given test case, takes source code in C#, builds it, instruments it, verifies that all required methods and types were instrumented, and executes it – testing that the CLR does not crash or kill the instrumented program due to an invalid code.

In order to create a functional test for SharpDetect, users must create a single C# source file with an entrypoint that starts the test. The source file should be placed inside the `Tests` folder, within the `SharpDetect.FunctionalTests` module. Moreover, it should be excluded from the build, but instead it needs to be copied to the output directory. Additionally, every test needs to be registered within the `TestDefinitions` class as a test case for its method `GenericTest`. An example of a test case definition using the `TestCaseAttribute` can be seen in the following code-listing. Additionally, each test needs to provide a path to a local configuration, or use the built-in property `UniversalConfig` that points to a basic configuration, which instruments everything inside the testing namespace.

```
[TestCase("Threading/LockKeyword.cs", UniversalConfig,
  TestName = "LockKeyword",
  Description = "Testing lock keyword injection")]
```

Furthermore, each test case may define a class that is responsible for checking that every tested construct was correctly identified and instrumented. The class needs to have the same name as the test but with the suffix `Checker` and derive from the abstract `BaseChecker` class. Users may define various `TypeRequirements`, `MethodRequirements` or `FieldRequirements` in the constructor of their checker. An example of how to add a new `TypeRequirement` can be seen in the following code listing.

```
var typeToInstrument = "TestedNamespace.TargetType";

Requirements.Add(new TypeRequirement()
  .AddEvent(AnalysisEventType.ClassCreate)
  .SetInjectedType(resolver.ResolveType(typeToInstrument)));
```

In the code-listing above, we can see a definition of a `TypeRequirement` that states that the class constructor of the type `TestedNamespace.TargetType` needs to be instrumented by injecting the class created analysis event. The implemented testing framework checks that all defined requirements are met during the instrumentation.

4.2.2 Continuous Integration Pipeline

Nowadays it is quite common to automate certain tasks connected with the development process, such as testing, artifacts generation, preparation of releases or even application deployment. While this is certainly convenient for developers, it provides also a way to avoid unnecessary mistakes compared to performing these tasks manually. Due to this reasons, we decided to implement a declarative continuous integration pipeline using the Gitlab API [26].

The pipeline is based on the `mcr.microsoft.com/dotnet/core/sdk:2.1` image, which is basically Debian 9 with pre-installed .NET Core SDK version 2.1. We configured the pipeline to run whenever someone pushes commits to the main development branch. The pipeline consists of the following stages:

- **Build**: this stage ensures that all SharpDetect modules, including the testing modules, can be successfully built
- **Test**: this stage is responsible for running unit tests and functional tests
- **Release**: this stage is responsible for generating persistent build artifacts that can be used to create releases. The Release stage is run only in case a new tag was created, or it can be executed manually.

Successful execution of the pipeline ensures that SharpDetect works also on Linux, apart from Windows where it was mainly developed. This as a result fulfils the goal **(G4)** about supporting multiple platforms, as described in Section 1.1.

4.2.3 SharpLab and DnSpy

In this subsection we present two very useful tools that were invaluable throughout our work in this project. The first of these tools is SharpLab [28], which is capable of showing results of code compilation – not only CIL, but also the resulting native code generated by the JIT compiler. Furthermore, SharpLab is currently capable also of running code, visualizing syntax trees, collecting profiling metrics and more.

The second tool we would like to mention is called DnSpy [29], which is created by the same author as the already mentioned library dnlib. DnSpy can be used as a hex editor, .NET assemblies editor and also a debugger. One of the most interesting features of DnSpy is the ability to debug .NET assemblies even without their original source code. This is quite useful when looking for errors in instrumented assemblies.

4.3 Development Diary

Throughout this section we present various problems that we encountered while designing and implementing SharpDetect. We describe the individual problems, as well as some solutions. While the vast majority of the problems was resolved during the work on this project, in few cases we only suggest possible solutions that are left for future work.

4.3.1 Instrumentation Routines

One of the first tasks was to design the instrumentation routines. These routines are responsible for generating CIL code that is capable of capturing information about analysis events and notifying SharpDetect. Two particularly interesting cases involved method calls and field accesses – both will be further discussed in the following paragraphs.

Method Calls and Returns

There are multiple ways to capture information about method calls and their returns. Two most notable approaches can be seen in the following code-listings.

```
1 public void CallerMethod() {
2   CalledMethod(...)
3 }
4
5 public void CalledMethod(...) {
6   /* Notify SharpDetect about method call */
7   /* Implementation */
8   /* Notify SharpDetect about method return */
9 }
```

We can see that in the first approach the analysis would rely solely on the instrumentation of the target callee method's body. The advantage of such approach is the ability to use CIL instructions `ldarg.*` for accessing individual arguments that simplifies their capturing. However, there are two major disadvantages. The first is that this approach does not work on methods without CIL bodies, for example when calling native code as described in Section 3.4.5. The second disadvantage is that the evaluation stack needs to be manually inspected for return values. Moreover, methods can have multiple return points, which needs to be accounted for as well.

```
1 public void CallerMethod() {
2   /* Notify SharpDetect about method call */
3   CalledMethod(...)
4   /* Notify SharpDetect about method return */
5 }
6
7 public void CalledMethod(...) {
8   /* Implementation */
9 }
```

The second approach is quite the opposite from the previous one. In this approach both events are instrumented in the caller method. The most important advantages of this approach are (i) that it works uniformly on all methods and (ii) the fact that the return value can be easily retrieved as it is the top-most item available on the evaluation stack. The disadvantage, however, is that a manual inspection of the stack needs to be performed in order to capture arguments of the method call. Similarly to obtaining the return value, all arguments must be present on the evaluation stack before the call instruction.

Due to the mentioned reasons, SharpDetect instruments method calls and their returns as described by the second approach. Even though both approaches have their disadvantages, the chosen approach works uniformly for all types of

method calls within managed code and as such provides better coverage compared to the first approach.

Field Accesses

Capturing information about field accesses might not seem problematic at the first glance. We already covered CIL programming with fields in Section 3.4.4. Nonetheless, there is an issue with capturing the target object when accessing instance fields whose declaring types are value types. But before we get to that, let us consider a simpler case when the declaring type is a reference type, which can be seen in the following code-listing.

```
1 public class RefType { public int Field; }
2
3 public void Method(RefType instance) {
4   var copy = instance.Field;
5 }
```

From the presented code-listing above, we are mostly interested in the method that loads an instance field to a local variable. In order for SharpDetect to capture information about this event, it needs to obtain information about the field, but also a reference to its target object. In the following code-listing, we present the disassembled implementation of the method in CIL.

```
1 IL_0000: ldarg.1
2 IL_0001: ldfld int32 RefType::Field
3 IL_0006: pop
4 IL_0007: ret
```

Based on the disassembled method code, we can see that it first pushes a reference to the `ReferenceType` instance onto the evaluation stack. This reference is obtained from the method's first argument using the instruction `ldarg.1`. After that, the field is loaded using the instruction `ldfld`. Clearly, storing the obtained reference as `System.Object` is a valid operation. Even in the case of heap compacting, as described in Section 3.1, the reference is still valid. In the following code-listing, we present a very similar example, however, a bit more complex field is being accessed.

```
1 public struct ValType { public int Value; }
2 public class RefType { public ValType Field; }
3
4 public void Method(RefType instance) {
5   var copy = instance.Field.Value;
6 }
```

In the code-listing above, we can see that there are actually two fields accessed: (i) using the provided instance of `RefType` the first field gets accessed and then, (ii) using the obtained `ValType` instance the second field gets accessed. Therefore, in order to access the value of the latter field, either a `ValType` or a managed pointer to a `ValType` needs to be pushed onto the evaluation stack. Generally, the second approach is preferred as can be seen in the following code-listing where we showcase the disassembled body of the method from the previous code-listing.

```
IL_0000: ldarg.1
IL_0001: ldflda valuetype ValType RefType::Field
IL_0006: ldfld int32 ValType::Value
IL_000b: pop
IL_000c: ret
```

From the previous code-listing, we can see that in order to access the `ValType` instance, the instruction `ldflda` is used. This instruction pushes onto the evaluation stack a managed pointer – basically a `System.IntPtr`. Using the managed pointer, the actual value is then retrieved and pushed onto the evaluation stack using the instruction `ldfld`.

Undoubtedly, storing information about the instance of `ValType` is harder than in the previous case. The `System.IntPtr` that gets pushed onto the evaluation stack is an unsafe pointer that points to a `ValType` instance, which is a value type that is not tracked by GC as described in Section 3.3.1. Due to this fact, SharpDetect can not store this pointer as we would not be able to determine whether it is still valid. Furthermore, if SharpDetect tried to load the value based on the managed pointer indirectly, boxing it, and then passing it as an instance of `System.Object`, the obtained reference would be meaningless since boxing creates a copy of the value as described in Section 3.4.6.

An important observation is that in .NET we can not have true global value types. In other words, all value types are transitively owned by a class or one of its instances. Therefore, in order to obtain information about accesses on fields that are defined on value types, SharpDetect should track its closest enclosing reference type. Current implementation of SharpDetect does not support this feature and during the instrumentation emits a warning.

4.3.2 Instrumenting Blocking Synchronization Actions

Once we implemented the basic `MethodCalled` and `MethodReturned` events, we naturally wanted to use these events to implement more advanced events as extensions to those previously defined, which was described in Section 4.1.5. Specifically, we started by implementing the `LockAcquired` and the `LockReleased` events as extensions to the `MethodReturned`, since these events were meant to trigger whenever the static methods `Enter` and `Exit` from the class `System.Threading.Monitor` returned. However, this design had one important flaw – analysis plugins could not easily tell that a thread is potentially blocked. Therefore, it would have been hard to implement, for example, a plugin that would analyse deadlocks.

Due to the problem that we just described, certain analysis events, such as the already mentioned `LockAcquired` event, were split into two events – (i) an event occurring before the actual action and (ii) an event occurring after the action completed. For example, instead of the `LockAcquired` event, we added events `LockAcquireAttempted` and `LockAcquireReturned`. The `LockAcquireAttempted` event is always injected before the critical section, just before the lock acquire action. Subsequently, the `LockAcquireReturned` is injected inside the critical section at its beginning. By implementing this solution, SharpDetect provides a more useful API to implement analysis plugins.

4.3.3 Restrictions on Instrumentation

Almost immediately after implementing the first instrumentation routines, we encountered another interesting problem. Regardless of the configuration provided by the users, `SharpDetect.Injector` must be aware that in some cases instrumentation should be skipped. Particularly, there are the following important cases to consider.

SharpDetect should not instrument itself

Based on the sequence diagram in Figure 4.2, we can see that such behaviour could quickly result in an endless recursion. Therefore, due to this reason, SharpDetect can not be analysed by itself and any attempt to do so must be ignored.

SharpDetect should not trigger analysis events

Similarly to analysed programs, SharpDetect is also a .NET program that even runs within the same process as the analysed program. Basically, this means that SharpDetect shares code with analysed programs. Therefore, it is important to distinguish events triggered by the analysed program from those that were triggered by SharpDetect, which just happens to use the same code.

Due to this problem, each event handler in `SharpDetect.Core` needs to check the thread-static flag `EventForwardingSupressed` that is defined in the class `EventDispatchingFrontend`. Moreover, each event handler needs to set this flag on its invocation and clear it just before it finishes. To simplify the process of setting and clearing the flag, SharpDetect introduces a custom `IDisposable` struct that can be used to set and clear the flags automatically.

```
1 using (new InternalSection()) {
2   /* 1. Preprocess event information */
3   /* 2. Send to analysis plugins */
4 }
```

In the code-listing above, we can see a simple solution to the mentioned problem. In order to ignore events triggered by SharpDetect, we must enclose event handlers using an instance of the `InternalSection`, whose constructor sets the required flag and its `Dispose` method clears it. Moreover, since `InternalSection` is a struct, there is no dynamic allocation involved and it only takes one byte on the stack because it does not have any fields.

4.3.4 Passing Live Objects

From the description of presented instrumentation routines, one can notice that SharpDetect passes live objects to analysis plugins. While this approach is necessary to track various information about the objects, there is a couple of important issues that users (e.g., authors of plugins) should be aware of when dealing with live objects from an analysed program.

- **State of objects might non-deterministically change.** However, it is still safe to perform some operations, such as comparing by reference. Moreover, if SharpDetect passes an argument or a return value that was originally of a value type, it must be boxed first, which creates a copy as

described in Section 3.4.6. Therefore, in this cases, as well as in other cases where the object is certainly immutable, it is safe to perform also many other operations.

- **Storing references might result in memory leaks.** Analysis plugins should either quickly throw away all references or hold them using weak pointers so that they do not affect GC. SharpDetect could otherwise suddenly require enormous amount of memory. Moreover, it is probably not that useful to perform analysis on objects that would be under normal circumstances already removed by GC.

Even though we mentioned that in the case of value types SharpDetect boxes obtained values, which effectively creates their copies, it never happens in the case of reference types. Otherwise it would be generally hard, for example, to group accesses to the same instance field if SharpDetect always created a copy of the instance.

4.3.5 Event Dispatching

An interesting challenge was to come up with a solution on how to raise analysis events. We wanted to use the publish-subscribe pattern, where the publisher would be an object that is accessible from any managed code and the subscriber would be `SharpDetect.Core`. The publisher would define a .NET event for each analysis event and a public trigger method so that events can be raised also by other classes. However, the most important question was where to inject the publisher.

It quickly became obvious that in order for the publisher to be accessible from any managed code, it needs to be implemented either (i) in an assembly which is a common dependency for all other assemblies or (ii) in a new assembly that would be injected as a new common dependency. Luckily, .NET Core and all other implementations have their own core library, which is a common dependency like in our case `System.Private.CoreLib`, as described in Section 3.7.

The other mentioned approach is in fact not possible. This was also covered in Section 3.7. Basically, the reason is that core libraries define basic types and contribute to the initialization of the CLR. Therefore, it would not make any sense for a core library to reference a managed assembly, as the assembly would need to use the types defined by the core library before the CLR had a chance to fully initialize them. The only way around it would be to implement the publisher using native code since there are no limitations on referencing native libraries.

Due to the mentioned reasons, SharpDetect injects the singleton class `EventDispatcherBackend` directly into the `System.Private.CoreLib`, together with individual events and helper trigger methods. The definition of these events is provided by `SharpDetect.Common` and they are subsequently instrumented by `SharpDetect.Injector`. Raised analysis events eventually need to be handled by `SharpDetect.Core` and then passed to analysis plugins. This was a high-level overview on how event dispatching can be achieved, but there are some more interesting problems that need to be resolved for event dispatching to work as described above. We discuss these problems in the following subsection.

4.3.6 Concurrent Invocation of Event Handlers

In Section 4.1.3, we mentioned that analysis events are always handled by the same thread that triggered them. Therefore, if the analysed program utilizes multiple threads, event handlers can be invoked concurrently. In this case there is also a possibility that some events might appear in a wrong order.

Consider the following example: thread `T1` releases a lock `L`. But right before SharpDetect notifies plugins about the release of lock `L`, a context switch happens – thread `T2` now runs instead of `T1`. Immediately after that, `T2` takes lock `L` and notifies plugins about this event. Therefore, in this case, plugins receive information about `T2` taking lock `L` before the notification from `T1` about the release of `L`. The solution to this problem is left as future work.

In order to resolve the mentioned issue, we need to impose a more strict ordering across the analysis events from multiple threads. One way to get a more strict ordering would be to implement some form of vector clock into the event dispatcher within the `SharpDetect.Core` module. This would generate a causal ordering, based on which SharpDetect could delay dispatching of certain events, such as in the example as described above, until all events that precede the current event are dispatched.

4.3.7 Modifying Core Libraries

We already know that in order to develop a dynamic analysis framework for .NET, the framework needs to be able to instrument also core libraries. Moreover, we already covered the importance and special properties of core libraries in the previous subsection and also in Section 3.7. Some of the mentioned facts make core libraries harder to instrument. In fact, during the work on this project we actually tried a simple proof-of-concept instrumentation of core libraries on all three major .NET implementations: .NET Framework, .NET Core and Mono. In the following paragraphs, we describe how we approached this task, what worked, as well as why we could not support one of the three cases.

.NET Frameworks

We start by describing the only case where we could not instrument core libraries in a way that would be usable by SharpDetect. The reason is that .NET Framework, the first predominant CLI implementation, introduced a concept called Global Assembly Cache (GAC), whose main purpose is to provide a machine-wide cache of assemblies. Furthermore, this cache is the first place where the CLR looks whenever it needs to load an assembly. All assemblies stored in GAC need to have strong names as described in Section 3.9.

However, when doing offline instrumentation, the mentioned assembly resolving mechanism is problematic. Anytime the CLR needs to load an assembly, it ignores other paths if the assembly exists in GAC. Therefore, if we instrumented an assembly whose original version exists in GAC, the instrumented assembly would be always ignored by the CLR. Even though it was already presented that the core library of the .NET Framework can be modified [30], all approaches relied on loading the instrumented core library into

GAC, which removes the original version and as such affects all .NET programs on the given machine.

Basically, the only approach to force the CLR to ignore GAC is to run programs with a setting called development installation that is not well documented nor supported. However, as some Microsoft developers mentioned on their blogs [31], this setting is used only for internal development purposes.

We can see that instrumentation of core libraries and even many other assemblies that are located in GAC is a hard problem in the case of the .NET Framework. Fortunately, the strong enforcement to prefer GAC when resolving assemblies is used only on the .NET Framework. In the following paragraphs we will show that GAC is no longer such a problem on the other tested CLI implementations.

Mono

Mono is an open-source implementation of the .NET Framework that also uses GAC. Even though the default assembly resolving mechanism is almost identical to the one used by the .NET Framework, Mono developers implemented a switch that makes it possible to bypass GAC. This can be achieved by setting the environment variable `MONO_PATH` to the directory that should be probed first when resolving referenced assemblies.

The solution described above works even in the case of core libraries. Compared to the .NET Framework's development installation, this setting is well supported and we did not encounter any problems while testing it.

.NET Core

In case of .NET Core, its development team chose not to implement GAC or to use any similar concept. The instrumentation process of core libraries and ensuring that they get loaded by the CLR is much simpler. Together with being able to even list the correct dependencies for any .NET Core program as described in Section 3.8, the instrumentation of analysed programs together with their dependencies is a significantly easier task.

Native Images

The only common problem for all mentioned implementations are native images. We already covered native images in Section 3.5.1, but basically they are assemblies that contain both CIL code and its corresponding native code. The CLR prefers execution of native code as it can be run without any interventions from a JIT compiler.

To ensure that the CLR actually loads instrumented versions of assemblies, SharpDetect needs to disable the use of native images. In .NET Core this can be achieved simply by setting the environment variable `COMPlus_ReadyToRun` to zero. However, even after this setting, the native code is still present if assemblies were generated using the CrossGen tool, as described in Section 3.5.1. Furthermore, if SharpDetect instrumented these assemblies, their native code would differ from their managed code, which can cause various problems. But based on a personal communication with the developer of dnlib, we decided to implement a suggested

work-around to completely strip the native code from instrumented assemblies. This ensures that resulting assemblies are still valid and do not contain any native code.

In order to strip all the native code, SharpDetect needs to perform a few modifications to the PE header prior to storing instrumented assemblies. PE header was already briefly introduced in Section 3.4.1. First, it has to set the `IsILOnly` flag and clear the `ILLibrary` flag, both of which can be found in the `Cor20` part of the PE header. Moreover, it needs to make sure that the `Machine` property for the given module is set to a valid architecture. Specifically, when setting the `Machine` property, only architectures that are not used for native images should be used. After SharpDetect successfully removes the native part of the assembly, it basically forces the JIT compiler to always compile types and methods of the given assembly, and therefore we can be sure that the instrumented code is actually executed.

Before the implementation of this work-around, we were not able to successfully run instrumented assemblies on other platforms than Windows. The execution always failed with a message *Failed to initialize CoreCLR*, where the CoreCLR is the CLR for .NET Core. Fortunately, by striping off native code of the instrumented assemblies, we were able to successfully run SharpDetect also on two additional platforms: Ubuntu 18.04 and Debian 9 (using the continuous integration pipeline described in Section 4.2.2).

4.4 Possible Improvements

Throughout this chapter we covered general overview of SharpDetect and described its individual modules together with some details about their implementation. Moreover, we discussed some interesting challenges that we faced together with approaches that we tried in order to solve them. Later, in the testing stage of this project, we discovered and identified certain aspects that can be improved. In the following subsections, we will cover these suggested improvements that are left as future work for this project.

4.4.1 Complexity of Programming in CIL

In Section 3.3 we presented an overview of programming using CIL. However, the language constructs shown within the section form by no means a comprehensive list. Actually, throughout the development we kept discovering new ways of expressing various operations that undoubtedly complicated the development of the `SharpDetect.Injector` module.

This problem mostly stems from the fact that there is a lack of proper tools that would simplify the process of development using CIL. For example, there is a utility called `PEVerify.exe` [32] that is shipped with .NET Framework installations and that is capable of verifying entire assemblies. The downside is that this tool does not support .NET Core and verifying core libraries. Therefore, the process of validating instrumented code can be really hard and error-prone. Despite the fact that there is a new tool called `ILVerify.dll` [33] that does not have the mentioned limitations, it is still in development.

Currently it seems that in order to make sure that SharpDetect produces correct code, a lot of testing is needed.

Due to the already mentioned facts, we can not currently guarantee that SharpDetect produces correct code for every possible input program. Despite this, we made sure that correct code is instrumented for all tests defined within the module `SharpDetect.FunctionalTests`. Further testing and code verification is still needed in order to gain better coverage of .NET constructs.

4.4.2 Observer effect

The term "observer effect", commonly known in quantum physics, is used to describe a phenomenon where any observation of an experiment may change its outcome. This is also relevant to SharpDetect because in order to observe the behaviour of a program, we need to instrument it first. Therefore, SharpDetect tries to reason about the behaviour of the original program based on its instrumented version.

One of the most important goals of analysis tools that rely on instrumentation is to make sure that they do not alter the behaviour of the program in a way that would interfere with the analysis. However, this is sometimes very hard to achieve. For example, consider the method in the following code-listing.

```
.method public hidebysig instance void Method () cil managed {
  .maxstack 8
  IL_0000: newobj instance void [System.Private.CoreLib]System
     .Object::.ctor()
  IL_0005: pop
  IL_0006: ret
}
```

The method in the code-listing above allocates a new instance of `System.Object`, pops its reference from the evaluation stack and returns. In the following code-listing we can see the disassembled version of the same method that was compiled into x86 assembly using the JIT compiler. Additionally, the original module was built in Debug mode that disables some JIT optimizations.

```
/* Disassembled method that was compiled by the JIT compiler
 *   The instruction newobj was compiled to lines 8 - 12
 *   Surrounding nops simplify debugger attaching */

Class.Method():
  // Truncated (8 instructions)
  L0022: nop
  L0023: mov rcx, 0x7ff9a5cd0ae8
  L002d: call 0x7ffa05834690
  L0032: mov [rbp-0x8], rax
  L0036: mov rcx, [rbp-0x8]
  L003a: call System.Object..ctor()
  L003f: nop
  // Truncated (3 instructions)
  L0046: ret
```

From the truncated x86 assembly code-listing above, we can see that a very simple method in CIL can be actually translated into many native instructions. We will not cover the entire presented code-listing – many of the instructions are there only to enable debugging anyway. However, the most important

instruction is on the line 12 with address `L003a` that actually calls the constructor of `System.Object`. Now in the following code-listing, we present the disassembled version of the same method, whose module was originally built in the Release mode. This configuration enables more advanced JIT optimizations.

```
1 C.Method()
2   L0000: ret
```

The resulting code created by the JIT compiler is quite different compared to the previous code-listing. Basically, the JIT compiler correctly deduced that the allocation in the method can be removed because the created instance is not used anywhere. Furthermore, it is not visible by other threads, the obtained reference is immediately thrown away and the specified default constructor does not have any side-effects.

We will now try to describe why is the presented behaviour, specifically JIT compiler optimizations, problematic for SharpDetect. Since we already described the instrumentation routine for object allocations in Section 3.4.2, we know that SharpDetect would generate an event in both presented cases. This is caused by the fact that SharpDetect generally knows nothing about optimizations that the JIT compiler is able to perform on the analysed code because SharpDetect operates on the bytecode level rather than on the native assembly level. Furthermore, when SharpDetect instruments the method by introducing the object created event, the JIT compiler is no longer able to perform the presented optimization because the reference to the newly created instance is not thrown away but instead it is passed to SharpDetect.

Even though the presented example is rather simple, its main aim is to give the reader a basic understanding of the problem. Moreover, the JIT compiler is capable of performing many optimizations, such as method inlining, removing unnecessary loads or copy propagations, all of which can in theory cause SharpDetect to introduce observer effects. In the following paragraph we present an option how we can minimize the impact of the observer effect on the analysed programs.

Minimizing Impact of the Observer Effect

When discussing different instrumentation approaches, we already mentioned the profiling API in Section 2.3.4, an approach for online instrumentation at the bytecode level. Even though this option was due to various reasons discarded, it could be particularly helpful when minimizing the impact of the observer effects.

The profiling API exposes a mechanism for direct observation of the CLR. The runtime itself is then capable of notifying registered profilers about certain events raised by the execution of the analysed program. The advantage is that these events can be configured to raise even without instrumenting anything. Furthermore, it is possible to configure the profiler to receive only subset of events, which could be used to prevent stack overflows when calling high-level languages from the profiler.

Even though all disadvantages that are mentioned in Section 2.3.4 are still valid, an ideal implementation of a dynamic analysis framework for .NET would probably use to some extent also the profiling API.

4.4.3 Analysis Performed by Managed Code

Majority of .NET programmers should try to stay within managed code as much as possible because it provides many advantages as described in Section 3.1. However, when programming an analysis tool for .NET programs that relies on bytecode instrumentation, in contrary it can introduce some problems. Actually there are three particularly problematic cases:

- As we already covered in Section 3.7, some types and methods from core libraries are needed during the initialization phase of the CLR. Instrumentation of these types and methods before the initialization is over may result in crashes.
- `SharpDetect.Core` and `SharpDetect.Plugins`, the most important runtime modules of the tool, are implemented using managed code. Therefore, users must be careful when instrumenting types and methods that are internally used by SharpDetect for analysis.
- There is not enough information exposed to the level of the managed code from the underlying CLR. For example, SharpDetect is only able to observe user-created threads, and not threads from the threadpool or other system threads like the GC thread or the finalizer thread.

We showed that when developing a dynamic analysis framework, usage of managed code can be in certain cases disadvantageous. In order to resolve this problem, it seems that the best course of action would be to introduce a small unmanaged SharpDetect module. Furthermore, if we integrated this module with the profiling API, we would gain the ability to instrument code online, listen to profiler events and also obtain more detailed information from the CLR about the analysed program's execution. SharpDetect would no longer fully rely on instrumentation, but instead in critical cases it could fallback to listening to profiler events in order to impact the analysed program in a less significant way.

4.4.4 Virtual Method Dispatching

In Section 3.4.5, we discussed the instruction `callvirt` that can be used to call virtual method. Basically, the actual invoked method is determined at runtime based on the target object instance.

Therefore, whenever SharpDetect instruments a method call to be invoked using the instruction `callvirt`, it is generally not possible to determine the actual method prior to the execution. This can be partially resolved by searching for all derived types from the declaring type of the method call receiver and instrumenting all of its overrides. However, SharpDetect can perform this only for all referenced assemblies. This means that if an assembly loads another assembly dynamically using, for example the method `System.Reflection.Assembly.LoadFrom`, SharpDetect may fail to find some overrides. Similarly to previous issues, this is exactly the place where an integration with the .NET profiling API would be quite useful. Using the profiling API we would obtain an information about the actual method that would be instrumented online.

4.4.5 Mapping Analysis Events to Original Source

Current implementation of SharpDetect maps individual analysis events to affected types, methods and fields based on metadata as described in Section 3.4.1. However, in some cases a more detailed information might be helpful, specifically a mapping of individual analysis events back to the original source code when possible.

When building .NET programs, it is common to generate also debugging information in the form of PDB files [34]. PDB is a proprietary file format developed by Microsoft for storing debugging information. Therefore PDB files, if available, could be used by SharpDetect to provide additional information, such as already mentioned mapping to original source code, or names of local variables. Moreover, this information could be quite helpful when working with bigger assemblies, or when integrating SharpDetect into an extensible development environment with the intention of directly marking source code lines with potential issues.

Actually, dnlib [14], the library SharpDetect uses for bytecode inspection and generation already supports PDB files. Therefore, information about the mapping of individual analysis events can be stored either separately to a file or using dnlib as an embedded resource to subject assemblies. Furthermore, analysis events can be assigned unique identifiers that might be, when necessary, queried in the prepared resource to obtain the debugging information. This solution would affect the execution of dynamic analysis only in a small way, as the debugging information can be queried by SharpDetect at the end of its execution.

5. Evaluation

In this chapter, we present an evaluation of the SharpDetect tool. We discuss some of its interesting performance aspects and implemented optimizations. Moreover, we discuss how much SharpDetect impacts analysed programs.

5.1 Performance Optimizations

As we previously indicated, dynamic analysis negatively affects runtime performance of analysed programs. While the overhead of dynamic analysis is generally inevitable, we dedicate this section to describing some approaches that were implemented in order to keep the overhead reasonable and practical.

5.1.1 Minimizing Dynamic Allocations

The CLR is generally capable of performing quite fast dynamic allocations due to the heap compacting mentioned in Section 3.1. As a result, memory allocation is as simple as increasing the pointer that points to the next free memory chunk on the heap. Despite this, frequent creation of objects with a short life span often creates unnecessary pressure on the memory management, as well as on GC that is triggered more frequently. These small objects are, in our case, mainly instances of analysis events.

We considered the following two possible solutions to tackle this issue: (i) introducing object-pools and (ii) using value types. Undoubtedly, both solutions solve the mentioned problems because they generate little to no pressure on the memory management and as a result, the overhead of GC is negligible. Moreover, the correct usage of value types can provide even greater performance advantages, since value types can be stored directly on the stack and therefore, all accesses on their fields are direct because there is no indirect reference to the heap.

Due to these reasons, all analysis events are created as `ValueTuple`s. When dispatching analysis events, SharpDetect performs dynamic allocation only when packing method arguments. This might be improved (i) either by introducing object-pools for arrays that store method arguments (ii) or by using the `stackalloc` operator to allocate arrays on stack, as well.

5.1.2 Avoiding Memory Leaks

For the purpose of dynamic analysis, SharpDetect needs to store information about some of the events. Since SharpDetect works with live objects, it needs to find out when are the objects released by the analysed program and release them as well, otherwise SharpDetect introduces potential memory leaks.

Since the CLR already uses GC to track live objects, SharpDetect can benefit from the same mechanism, too. For example, when tracking accesses to instance fields, SharpDetect can drop all records once their owner instance is collected by GC. In order to implement this functionality, we can make use of the `System.WeakReference` type or collections built using weak references, such

as the `System.Runtime.CompilerServices.ConditionalWeakTable<K,V>`. Once the last non-weak reference is released, GC is free to collect the object. This ensures that SharpDetect does not unnecessarily hijack unused memory and thus does not create memory leaks. `ConditionalWeakTable`s are mostly used by plugins that keep track of accessed memory locations.

5.1.3 String Interning and Caching Resolved Identifiers

In order to identify methods and types that triggered individual analysis events, `SharpDetect.Injector` packs the full method identifier as a string. Right before an event is dispatched to analysis plugins, the string gets parsed into a descriptor that is easier to work with in analysis plugins.

Furthermore, an important observation is that the string identifiers are injected during the instrumentation process and therefore appear in the CIL as string literals. The library that SharpDetect uses for instrumentation, dnlib [14], behaves in this case similarly to C# compilers – effectively interning string literals. This ensures two useful things at runtime: (i) for each interned string there exists only one instance, (ii) when comparing interned strings, it is sufficient to compare just their references. The obvious downside is a larger `#Strings` section within the PE header of instrumented assemblies. We already covered PE headers in Section 3.4.1. However, the undeniable advantage of interning the string identifiers is the fact that comparison of interned strings is generally very fast.

In order to provide more useful environment for the development of analysis plugins for SharpDetect, we mentioned that identifiers of methods are parsed into instances of the `MethodDescriptor`, similarly their declaring types are parsed into `TypeDescriptor`s and so forth. These descriptors are cached and their instances are shared across all occurrences. Therefore, when developing analysis plugins, SharpDetect users may rely on the fact that whenever a specific method is called, the same `MethodDescriptor` is passed to the plugin. Analogically, the same rule applies to all other events as each type, field, method and parameter is parsed only once and the obtained instance is reused for subsequent events.

5.2 Measurements

In this section, we report and discuss the results of measurements of time and memory overhead of SharpDetect on the selected subject programs, which we performed in order to estimate how much usage of the tool impacts analysed programs. We begin by describing the environment under which the measurements were executed together with the methodology of measuring the running time and memory consumption. After that, we present the individual subject programs, as well as the results of time and memory measurements.

5.2.1 Environment

For the purpose of measuring overhead of SharpDetect, we used the following hardware and software configuration.

- **CPU**: Intel Core i7-8550U @ 1.80GHz
  - **Cores/Threads**: 4/8
  - **Cache**: 4 x (32 + 32 + 256) + 8MB
- **Memory**: 16GB
- **Operating System**: Windows 10, 64bit version
- **.NET Core**: 2.1

5.2.2 Measuring Time Overhead

In order to measure the time overhead of SharpDetect, we used the powershell utility `Measure-Command` that measures wall-clock. To obtain the baseline, we used this utility on the subject assembly prior to the instrumentation. Then, we continued by using the utility on the subject assembly after the instrumentation.

5.2.3 Measuring Memory Overhead

At the time of writing this thesis, there were no freely available profilers for .NET Core that are capable of tracing detailed information about the managed heap. Due to this reason, for the purpose of the memory measurements, we added a simple memory measuring routine that triggers after each dispatched analysis event. This routine obtains the current memory usage with the following method.

```
1 long System.GC.GetTotalMemory(bool forceFullCollection);
```

The described approach using the method above can be used to obtain the current memory usage on the garbage collected heap in bytes. For the purpose of these measurements, we did not force GC as we did not want to further impact the execution of the subject programs.

To obtain the baseline, which is equal to the memory usage on the subject program before the instrumentation, one may use the same measuring routine and execute it periodically inside a long-term task. Undoubtedly, this approach makes the subject programs use even more memory. However, compared to the rest of the program, this memory usage is quite small.

5.2.4 Subject 1: Simple Task Parallel Library Program

The first subject program uses Task Parallel Library (TPL) to process a big array in an unsafe way. More specifically, the possibility of a data-race is there almost for each array element access. Therefore, we expected the `Eraser` plugin to report plenty of violations, compared to the `FastTrack` plugin, whose total number of reports depends on the interleaving of individual threads. The subject program for this test can be seen in the following code-listing. Is is also available in the `SharpDetect/Examples/Evaluation1` directory of the electronic attachment.

```
using System;
using System.Threading.Tasks;

namespace Evaluation1 {

  class Program {
    static int[] Data = new int[10000];

    static void Main(string[] args) {
      for (var i = 0; i < Data.Length; i++)
        Data[i] = i;

      Parallel.For(0, Data.Length - 1,
        (index) => Data[index + 1] = Data[index] + 1);
    }
  }
}
```

Measuring Time and Memory Overhead

In this paragraph, we focus on the time and memory overhead of SharpDetect for the subject program presented at the beginning of this section. In order to cover the significant aspects that contribute to the time and memory overhead, we begin by presenting the original sizes of subject assemblies, as well as their sizes after the instrumentation, which can be seen in Table 5.1.

Input Assemblies		**Results**
Name	**Original Size [KiB]**	**Instrumented Size [KiB]**
Subject Assembly	4	7
Core Library	12 721	2 700

Table 5.1: Difference in sizes of assemblies before and after the instrumentation. Subject Assembly corresponds to the program from the beginning of Section 5.2.4. Core Library corresponds to the `System.Private.CoreLib` that is referenced by the Subject Assembly.

In Table 5.1, we can see that the instrumented subject assembly is about 1.75-times bigger compared to its original size. Furthermore, the instrumented core library is about 4.7-times smaller than its original size. This is due to the fact that SharpDetect strips native code off assemblies, as described in Section 4.3.7. As a result, the absence of AOT compiled native code will force the JIT compiler to compile also the core library at runtime, which is not normally required. Therefore, the overhead of the JIT compiler will be bigger, since the subject assembly is bigger and also the core library needs to be compiled, too.

Moreover, during the initialization phase, SharpDetect uses reflection, which has a constant time overhead that contributes to the total execution time, as well. The exact running time and memory consumption of the subject program under different configurations can be seen in Table 5.2.

Test Description		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 5.2: The running time and memory consumption of the subject program presented in Section 5.2.4. The first line provides the baseline, which stands for the subject program before the instrumentation. In the following lines, we present the running time and memory consumption of the instrumented subject program that was measured together with different plugins configuration.

In Table 5.2, we can observe that SharpDetect was responsible for a slow-down, which caused the subject program to run about 4-times slower compared to the baseline time. This slow-down was measured while using the `Eraser` plugin, which reported a data-race for almost each array element. The bigger memory overhead is caused by the fact that each array element is treated as an independent variable for which some analysis information needs to be tracked.

Furthermore, each measurement was repeated 50-times. In Table 5.2, we present average values together with their corresponding standard deviations. The measurements of the memory baseline did not deviate at all. This is due to the fact that at runtime, there is not a lot of dynamic allocation for the non-instrumented version of the subject program.

5.2.5 Subject 2: Producer-Consumer Program

The second subject program is a simple implementation of the producer-consumer pattern. In this implementation, both the producer and the consumer are implemented as separate `Task`s and share a common `Queue<int>`. The truncated version of the program can be seen in the following code-listings.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Threading;
4  using System.Threading.Tasks;
5
6  namespace Evaluation2 {
7    public class Program {
8      const int size = 100;
9      static Queue<int> queue;
10     static object lockObj;
11
12     public static void Main(string[] args) {
13       queue = new Queue<int>(size);
14       lockObj = new object();
15
16       var producer = Task.Run(() => Producer());
17       var consumer = Task.Run(() => Consumer());
18       Task.WaitAll(producer, consumer);
19     }
```

The code-listing above was truncated and its complete version can be seen in the `SharpDetect/Examples/Evaluation2` directory, which is a part of the electronic attachment. Basically, the producer task iteratively produces about 10 000 items for the consumer task to consume. In order to enqueue or dequeue an item from the queue, a lock needs to be acquired. Furthermore, this implementation uses signals where the producer task signalizes whenever an item is produced and waits for a signalization when the queue is full. Analogically, the consumer task signalizes whenever an item is consumed and waits for a signalization when the queue is empty. The producer task terminates by producing a "poison pill", which terminates also the consumer task once it is consumed.

Despite the fact that there are no data-races in this implementation, it is useful when analysing the overhead of SharpDetect because the program generates a lot of analysis events.

Measuring Time and Memory Overhead

Similarly to the previous subject program, we begin by presenting size differences of assemblies before and after the instrumentation. This can be seen in Table 5.3.

Input Assemblies		**Results**
Name	**Original Size [KiB]**	**Instrumented Size [KiB]**
Subject Assembly	5.0	10.5
Core Library	12721	2701

Table 5.3: Difference in sizes of assemblies before and after the instrumentation. Subject Assembly corresponds to the program from the beginning of Section 5.2.5. Core Library corresponds to the `System.Private.CoreLib` that is referenced by the Subject Assembly.

In Table 5.3, we can see that the instrumented subject assembly more than doubled in size compared to its original version. Furthermore, the instrumented core library is about 4.7-times smaller compared to its original version. This is due to the same reason that is described in the previous measurement in Section 5.2.4. The exact running time and memory consumption of the subject program under different configurations can be seen in Table 5.4.

Test Description		**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 5.4: The running time and memory consumption of the subject program presented in Section 5.2.5. The first line provides the baseline, which stands for the subject program before the instrumentation. In the following lines, we present the running time and memory consumption of the instrumented subject program that was measured together with different plugins configuration.

In Table 5.4, we can observe that the subject program was at most slowed down by about 3.7-times. Even though the memory usage seems to be about 16-times larger, we can see that there is a constant memory overhead of about 4000 KiB, regardless of the plugin configuration. This is caused by the fact that SharpDetect needs to track a lot of information during the execution, such as method arguments and return values. Therefore, even when the plugins do not need to track much of the data, it may still consume a significant amount of memory compared to the memory baseline.

Similarly to the first subject program, each measurement was repeated 50-times. In the Table 5.4, we present average values together with their corresponding standard deviations. Moreover, the measurements of the memory baseline did not deviate at all since during the runtime there is not a lot of dynamic allocation happening.

5.2.6 Summary

Throughout this chapter we presented multiple approaches that were implemented for the sole purpose of minimizing runtime overhead of SharpDetect. Furthermore, we analysed its time and memory overhead based on the selected subject programs. According to our measurements and the implemented performance solutions, we claim that SharpDetect minimizes its impact on the analysed programs, thus fulfilling the goal **(G3)** about performance as described in Section 1.1.

6. Using SharpDetect

This chapter is dedicated to a more detailed description of how to use SharpDetect. We cover various tasks – preparing a subject program and configuration, instrumenting assemblies and running the dynamic analysis. Furthermore, we also describe how to verify the generated code by SharpDetect, as well as the implementation of analysis plugins.

6.1 Preparing Subject Program

In order to illustrate the process of preparation and the subsequent execution of the dynamic analysis, let us consider the following simple program written in C#. This program is a part of the electronic attachment and can be found in the `SharpDetect/Examples/Sample` directory.

```
1  using System;
2  using System.Threading.Tasks;
3
4  namespace Sample {
5
6    class Program {
7      static int[] Data = new int[10];
8
9      static void Main(string[] args) {
10       for (var i = 0; i < Data.Length; i++)
11         Data[i] = i;
12
13       Parallel.For(0, Data.Length - 1,
14         (index) => Data[index + 1] = Data[index] + 1);
15     }
16   }
17 }
```

In the following sections, we will describe how to prepare the configuration and how to search for concurrency errors within the call to Task Parallel Library (TPL) [35] that can be seen in the previous code-listing. But first, we create a C# .NET Core console application project called ”Example” using the source code from the previous code-listing.

6.2 Preparing Configuration

In order to analyse the subject program, we need to make sure that required injectors are not disabled within any of the configurations – we are using mainly array, field and method injectors – these settings were presented in Section 4.1.1. Furthermore, we also need to set `MethodPatterns` and `FieldPatterns` in order to specify what exactly should be instrumented. Individual patterns represent paths to methods and fields – each pattern consists of a namespace, a declaring type and a name of the field or method delimited with double colons `"::"`. Furthermore, if users want to, for example inject all methods of a declaring field, they can omit the method name. `SharpDetect.Injector` compares these patterns during the instrumentation in

a full-text mode. Therefore, in order to analyse the call to TPL within the subject program, we can use the following local configuration.

```
{
  "TargetAssembly": "Sample.dll",
  "FieldPatterns": [ "Sample" ],
  "MethodPatterns": [ "Sample" ],
}
```

Using the local configuration above, we can instruct SharpDetect to instrument all analysis events within the `Sample` namespace. This configuration should be saved right next to the C# project file with name "Example.json".

6.2.1 Additional Configuration Settings

In Code-Listing 4.1, we presented the default content of the global configuration file. While the presented settings where already covered within Section 4.1.1, there are actually two more settings that can be quite helpful when working with SharpDetect but require a deeper understanding of SharpDetect and the CLR.

EnableJitOptimizations

Whenever we set the `EnableJitOptimizations` setting to true, it allows optimizations that are normally performed by the JIT compiler. We already briefly introduced JIT optimizations in Section 3.5.3, as well as, described how it can cause problems when analysing the behaviour of programs in Section 4.4.2. SharpDetect by default disables JIT optimizations using the environment variable `COMPlus_JITMinOpts`.

However, when performing dynamic analysis, it can be useful to run the analysis both with JIT optimizations and without them. The difference can be illustrated on the example program we presented at the beginning of Section 6.1. If we allowed JIT optimizations, we could observe that each execution of the provided lambda function is performed by the same thread, regardless of the fact that we used TPL and some users might expect it to execute in multiple threads. Analogically, if we turned the optimizations off, we could observe that each execution of the lambda function is performed by a different thread. Therefore, based on what we are planning to analyse, it might be useful to keep this setting in mind when executing dynamic analysis and evaluating obtained results.

For the purpose of this example, we will consider that JIT optimizations are turned off, as we want to showcase the capabilities of implemented plugins when searching for concurrency issues, which would not be possible if all the code is executed in the same thread. This setting can be defined on the command-line right before the execution of dynamic analysis.

VerifyInstrumentation

In Section 4.4.1, we introduced a tool called ILVerify [33] that can be used to verify .NET assemblies. Despite the fact that at the time of writing this thesis

the tool is still under development and not yet available as a stand-alone package, its integration into SharpDetect can be really useful.

The verification of the instrumented assemblies is performed, if the setting `VerifyInstrumentation` is set to true. Discovered violations found by ILVerify are printed to console with logging level set to `Warning`. The verification of `System.Private.CoreLib` is turned off because it generates many violations due to many unverifiable constructs used there. Despite this, throughout the development we made sure that types and methods instrumented into this assembly can be verified without any problems.

However, the integration of ILVerify into SharpDetect is experimental and the evaluation of verification results is not automatized. Basically, this can be a useful feature when implementing new features or fixing errors in `SharpDetect.Injector` where users immediately see whether the code that SharpDetect generates is verifiable or not. Needless to say, unverifiable code does not imply the code is not correct – for example core libraries use many unverifiable constructs. However, SharpDetect generally should not introduce new violations to the code.

For the purpose of this example, we will consider that the verification of instrumented assemblies is turned on, as we intend to showcase the output of the tool after the verification in the following section. This setting can be defined using the configuration files.

6.3 Generating Self-Contained Package

After we prepared the subject program and the necessary configuration, we can proceed by generating the self-contained package as described in Section 3.8 using the following command:

```
dotnet SharpDetect.Console.dll build <Example.csproj> \
      --rid <RID>
```

The RID from the previous command stands for a Runtime Identifier. The whole list of supported runtime identifiers (RIDs) can be found in the official documentation hosted by Microsoft [36]. This identifier needs to be specified in order to create a fully self-contained package.

6.4 Assemblies Instrumentation

Since we have the subject assembly prepared, as well as the configuration, we can run the instrumentation process using the following command. We intentionally run the command with increased verbosity as we intend to describe certain phases of the process.

```
dotnet SharpDetect.Console.dll instrument \
      <_SharpDetect\\Example.json> \
      --level Information
```

First, SharpDetect checks whether the target assembly is valid – it must have

an entrypoint, i.e. it must be executable, as well as, it must target the .NET Core framework. After the initial checks, SharpDetect begins the instrumentation process. For each analysis event it prints a message in the following format:

```
// HH:mm:ss        - 24-hour Timestamp
// Level           - Logging level
// Injector        - Name of the injector
// Instruction     - IL_xxxx: OpCode <Operand>
HH:mm:ss [Level] [Injector] Instruction
```

Based on the description above, after executing the mentioned command, we can expect an output similar to the following:

```
// SharpDetect loaded Example.dll targeting .NET Core 2.1
16:49:40 [INF] Assembly Sample, Version=1.0.0.0, Culture=
    neutral, PublicKeyToken=null targets framework .NETCoreApp,
     version 2.1.0.0
// First analysis event instrumented -- load of the Data array
16:49:42 [INF] [FieldRead] IL_0005: ldsfld System.Int32[]
    Sample.Program::Data
  at System.Void Sample.Program::Main(System.String[])
// Truncated (17 other analysis events)
```

After the successful instrumentation of the required analysis events, SharpDetect continues by storing modified assemblies and in our case verifying them, as well. During this phase, we can see an output similar to the following:

```
// Truncated (write of System.Private.CoreLib.dll)
// Writing instrumented assembly Sample.dll
16:49:44 [INF] Writing assembly Sample, Version=1.0.0.0,
    Culture=neutral, PublicKeyToken=null
// Running ILVerify on instrumented Sample.dll
16:49:45 [INF] ILVerify starting verification of <_SharpDetect
    \Sample.dll>
// No violations found in the instrumented assembly
16:49:45 [INF] Successfully verified all types and methods of
    the assembly
```

After the whole command finished, we can observe that SharpDetect created various analysis events. Moreover, for the purpose of dynamic analysis it needed to instrument both the `Sample.dll` and the `System.Private.CoreLib.dll` and it also verified that the `Sample.dll` should be safe to execute. Finally, since everything is ready we may proceed to executing the dynamic analysis, which is further covered in the following section.

6.5 Executing Dynamic Analysis

In the previous sections we described the process of creating a subject assembly based on the program presented in Section 6.1, we discussed the configuration – both for instrumentation and runtime, and finally we prepared instrumented assemblies for the dynamic analysis. This section is dedicated to executing the dynamic analysis and evaluating the obtained results. We can start by executing the dynamic analysis using the following command.

```
dotnet SharpDetect.Console.dll run \
  <_SharpDetect\\Example.dll> \
  --level Information \
  --config Eraser|EchoPlugin \
  --enableJitOptimizations false
```

Immediately after executing the command above, we can observe the initialization phase of SharpDetect, whose output is similar to the following.

```
// SharpDetect searches for plugins
19:36:50 [INF] Searching for assemblies in C:\SharpDetect\
    Plugins...
// Truncated (3 assemblies; searching for plugins...)
// SharpDetect found assembly with implementation of plugins
19:36:50 [INF] Found plugin EchoPlugin in SharpDetect.Plugins,
     Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
// Truncated (3 other plugins: Eraser, FastTrack, EmptyPlugin)
// Information about constructing plugin chain
19:36:50 [INF] Required plugins for analysis: ["Eraser", "
    EchoPlugin"]
19:36:50 [INF] Dynamic analysis framework successfully
    initialized.
```

We can see that SharpDetect starts looking for plugins in the directory `C:\SharpDetect\Plugins`, which SharpDetect obtained from the environment variable `SHARPDETECT_PLUGINS`. The process of searching for and initializing plugins is covered in Section 4.1.1. Furthermore, we can observe that after SharpDetect found all available plugins, it tries to construct a plugin chain based on the provided configuration. If the previous steps succeeded, SharpDetect is correctly initialized and the execution continues with the entrypoint of the analysed program.

At this point, `SharpDetect.Core` is dispatching analysis events that are being processed both by the `Eraser` and the `EchoPlugin`. We can observe a textual representation of individual analysis events provided by the `EchoPlugin` in the following format:

```
// HH:mm:ss        - 24-hour Timestamp
// Level           - Logging level
// ThreadID        - Managed thread ID
// EventCategory   - Event group (Field, Method, Array...)
HH:mm:ss [Level] [ThreadID] EventCategory: Description message
```

Furthermore, implemented analysis plugin use a slightly different format when reporting warnings and errors. This is because they usually do not report issues based on a single event but instead based on a sequence of events, therefore, some information categories, such as thread id do not apply to them. In order to report warnings and errors, analysis plugins use the following format:

```
// HH:mm:ss        - 24-hour Timestamp
// Level           - Logging level
// PluginName      - Name of the analysis plugin
HH:mm:ss [Level] [PluginName] Description message
```

Since we just showed the format of messages that are used by individual plugins, it should not be a big surprise that the configured plugin chain starts generating output similar to the following. Note that timestamps were removed and the array descriptor was shortened in the example below.

```
[INF] [1] Array: System.Int32[] created in method System.Void
    Sample.Program::.cctor()
// Truncated (up to the point the array is being filled)
[INF] [1] Array: System.Int32[] wrote value 0 on index 0 in
    method System.Void Sample.Program::Main(System.String[]).
// Truncated (loading array from field)
[INF] [1] Array: System.Int32[] wrote value 1 on index 1 in
    method System.Void Sample.Program::Main(System.String[]).
// Truncated (up to the TPL call)
[INF] [6] Array: System.Int32[] wrote value 4 on index 4 in
    method System.Void Sample.Program/<?>::<Main>b(System.Int32
    ).
// Found a possible data-race
[ERR] [Eraser] detected data-race on array element <array>[4]
// Truncated (the rest of the analysis)
```

From the output above, we can see that the `Eraser` plugin found a data-race. In fact, there is a data-race on each array element except for the element with the index 0. The `Eraser` plugin actually reveals the problem for each affected array element, but, for the purposes of clarity, the example output above was truncated.

6.6 Implementing Analysis Plugins

In Section 4.1.2, we already covered the basic information needed to implement a custom analysis plugin for SharpDetect. This section is dedicated to the guidelines that need to be obeyed in order for the created analysis plugins to work as intended. Furthermore, we discuss the implementation of the two non-trivial example plugins, the `Eraser` plugin and the `FastTrack` plugin, which were implemented as a part of this project.

6.6.1 General Guidelines

There are couple of rules and advices that users of SharpDetect need to keep in mind when implementing custom plugins or changing implementations of existing plugins. These guidelines can be expressed in the following points.

- **Threading model**: Every analysis event needs to be handled synchronously by the thread that raised the event. SharpDetect has no mechanism to distinguish custom analysis threads from the threads used by the analysed programs. Furthermore, analysis events may be raised concurrently – therefore, a proper thread synchronization might be needed when implementing some plugins.
- **Hijacking threads**: Even though in Section 4.1.3 we described that threads are hijacked in order to analyse the events they triggered, it is probably not a good idea to hold on these threads for extended periods of time. Clearly, by holding a thread, SharpDetect actually pauses execution of the analysed program. Therefore, all threads should be released immediately after the necessary analysis was performed.

- **Reference as few as possible**: In Section 4.4.3, we already covered issues in connection with analysing managed code using the same managed code. Generally, we want to minimize the number of referenced types and methods by analysis plugins in case users decide to instrument them. As long as users obey the first mentioned rule, SharpDetect has a mechanism to distinguish analysis events triggered by SharpDetect code and plugins from those that are triggered by the analysed program. But still it is advised to keep the plugin implementations simple.

Now that we covered the basic guidelines, together with the description in Section 4.1.2, users should have enough information to be able to write their own analysis plugins. In the following subsections, we focus on the description of the implementation of two analysis plugins that can be used to search for concurrency issues in analysed programs.

6.6.2 Eraser

The implementation of the `Eraser` [7] plugin can be found within the `SharpDetect.Plugins` module, more specifically in its subnamespace `LockSet`. In order to search for data-races, the plugin needs to keep track of the following variables that represent individual memory accesses:

- **Static fields**: The plugin needs to track information about the field only, since it is not bound to an instance. Furthermore, it needs to check whether the field is not marked as `[ThreadStatic]` because that would mean that each thread accesses its own instance and data-races would not be possible.
- **Instance fields**: The plugin needs to additionally track information about the instance the field belongs to. Otherwise, the plugin would report data-races also in case we accessed the same field but on different instances.
- **Array elements**: The plugin needs to also track accesses to array elements. These accesses consist of references to arrays and information about their specific elements in the form of indices.

According to Savage et al. [7], for each variable the plugin additionally needs to track a collection of candidate locks. This collection consists of locks that previously potentially guarded given variable access. Therefore, on each variable access, a lock refinement operation needs to be performed – generating an intersection of the locks currently held by the thread with the previous candidate locks. Whenever the plugin finds a variable whose resulting candidate locks collection is empty and its state is marked as shared-modified, it reports a possible data-race.

6.6.3 FastTrack

`FastTrack` [8] is a precise vector clock based data-race detector. Its implementation can be found within the `SharpDetect.Plugins` module, in the `VectorClock` subnamespace. Similarly to the `Eraser` plugin, in order to search for data-races, this plugin needs to keep track of memory accesses as described

in the previous subsection. Moreover, `FastTrack` monitors also these types of analysis events:

- **Thread fork**: The plugin needs to track information about creating and starting threads. As described in Section 2.1, SharpDetect currently supports this information only for the user-defined threads. This is due to the fact that there is almost no information available about forks of other threads, such as threads from `ThreadPool`, from the managed code.
- **Thread join**: Similarly to the thread fork event, this plugin needs to track also information about joining threads. However, the same limitations apply as with the thread fork event.

`FastTrack` uses an improved vector clock based algorithm for detecting data-races that adaptively switches between simple epochs to vector clocks when possible without the loss of precision. This technique effectively reduces the overhead of dynamic analysis compared to the traditional vector clock based data-race detectors. Furthermore, compared to the `Eraser`, this algorithm does not try to predict data-races and only detects them, but also does not produce any false-positives.

7. Case Study

In this chapter, we describe our experience with the application of the developed tool SharpDetect on an existing .NET library in order to search for possible concurrency issues. The tested library is called NetMQ [37], which is a fully managed C# implementation of ZeroMQ – a high-performance asynchronous messaging middleware that operates without a message broker server.

Independently from the development of SharpDetect, we observed a possible timing issue in NetMQ that occurred very rarely. In the following sections we describe how we used SharpDetect to search for the source of the timing issue.

7.1 Preparing for Dynamic Analysis

We already covered how to use SharpDetect in Chapter 6 and that in order to perform a dynamic analysis, the user needs to provide the following: (i) a subject program and (ii) a local configuration that describes the analysis. The resulting folder structure looks as follows:

```
.
|-- Worskspace
|   |-- CaseStudy
|   |   |-- CaseStudy.csproj
|   |   |-- CaseStudy.json
|   |   |-- Program.cs
```

The first step is to create a subject program. We can implement simple `ResponseSocket` and `RequestSocket` that represent a server and a client, respectively. The server is then bound to a localhost IP address on a fixed port and client connects to it. Moreover, both the server and the client are implemented to run in separate threads. Their subsequent communication is performed using two extension methods `SendMultipartMessage` and `TryReceiveMultipartMessage` that take `System.Timespan` instances as arguments in order to define timeouts. The described test should be implemented as a standard .NET Core console application. According to the folder structure described above, the project is called `CaseStudy` and its implementation is provided by the file `Program.cs`. A complete example that is used in this section can be found in the electronic attachment of this thesis in the `SharpDetect/Examples/CaseStudy` directory.

The second step is configuration, we need to configure what SharpDetect needs to instrument. In fact, this step is quite easy because we do not need to set any method patterns – locks are instrumented by default and for fields we can just set the library's common namespace, which is "NetMQ". Moreover, the configuration file should be placed right next to the C# project file and named `CaseStudy.json`. This configuration is part of the attachment as well, and its content can be seen in the following code-listing.

```
{
  "TargetAssembly": "CaseStudy.dll",
  "FieldPatterns": [ "NetMQ" ],
  "MethodPatterns": [ ],
}
```

Now we need to prepare the target assembly by creating a self-contained package as described in Section 3.8. In order to create a self-contained package, we need to provide a runtime identifier (RID) [36]. We assume that we are using the 64-bit version of Windows 10, whose runtime identifier is `win10-x64`. Then, we can generate the self-contained package using the following command.

```
dotnet SharpDetect.Console.dll build <CaseStudy.csproj> \
      --rid win10-x64
```

Without specifying the `--output` option as described in Section 4.1.1, the default folder is called `_SharpDetect`. The resulting folder structure is in this case the following:

```
.
|-- Worskspace
|   |-- NetMQ_Test
|   |   |-- _SharpDetect
|   |   |   | <self-contained-package>
|   |   |   | <SharpDetect-assemblies>
|   |   |   | <copy of CaseStudy.json>
|   |   |-- bin
|   |   |-- obj
|   |   |-- CaseStudy.csproj
|   |   |-- CaseStudy.json
|   |   |-- Program.cs
```

Now everything is prepared for the instrumentation process, which can be executed using the following command:

```
dotnet SharpDetect.Console.dll instrument \
      <_SharpDetect\\CaseStudy.json>
```

Finally, we are able to execute the dynamic analysis. In order to start it, we need to provide a list of analysis plugins that should be used for this specific run. We assume usage of the following two plugins: the `Eraser`, which reports possible data-race, and the `EchoPlugin`, which displays textual representation of individual analysis events. We can start the dynamic analysis using the following command.

```
dotnet SharpDetect.Console.dll run \
      <_SharpDetect\\CaseStudy.dll> \
      --config Eraser|EchoPlugin
```

Results obtained by running the dynamic analysis are described in the following section.

7.2 Evaluating Obtained Results

In the previous section, we covered the process of preparing the subject program and configuration, creating self-contained assemblies, instrumenting target assemblies and executing the dynamic analysis. Since the process of observing the behaviour of analysed programs was already covered in Section 6.5, we continue by evaluating obtained results.

As we are looking for errors, we can set the logging level to a more strict value, such as `Warning` or `Error`. Then, for the subject program and configuration we prepared, we can observe the following issues found by the `Eraser` plugin:

```
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
```

In the presented output above, we can see that the `Eraser` plugin found a possible data-race on the static fields `s_lastTsc` and `s_lastTime` defined by the class `NetMQ.Core.Utils.Clock`. If we look at the implementation of NetMQ, we can observe that the two mentioned static fields are read and written only by the following method [38]:

```
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 */
}
```

Now if we re-run the dynamic analysis with the logging level set to `Information`, we can see from the output below that the method is actually executed by multiple threads without any synchronization of the critical section.

```
// 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.
```

The basic format of the output above was covered in Section 6.5. However,

the description messages are custom for each event type. In case of field events, it consists of the following items:

- **Field metadata token**: describes the affected field. More specifically, it consists of its type, namespace, declaring type and name of the field. In the output above, both fields are of type `System.Int64`, their namespace is `NetMQ.Core.Utils`, their declaring class is `Clock` and their names are `s_lastTime` and `s_lastTsc`.

- **Operation**: describes the event type, based on which, we concatenate the field metadata token with either `" was read from."`, or with `" was written"`. In the second case, which corresponds to the FieldWrite event, the description message contains also the third item.

- **Written value**: is present only for the FieldWrite event. In this case we add string `" with value "` and concatenate it with the result of calling the `ToString()` method on the written object or value.

Moreover, at the beginning of this chapter, we mentioned that NetMQ is a C# implementation of ZeroMQ. If we looked at some other implementations, for example libzmq [39] implementation in C++, we would find that they actually use a mutex for thread synchronization of the presented method `NowMs`.

8. Conclusion

In this project, we designed and implemented a dynamic analysis framework that is capable of analysing the behaviour of .NET programs. Throughout the individual chapters of this thesis, we compared various related tools and environments, provided an introduction to the CLI, and discussed many aspects of .NET Core and its runtime – the CoreCLR. Moreover, we documented the tool that we created, SharpDetect, together with the development process, and described a couple of interesting problems that we encountered during our work on this tool. Finally, we discussed how to use and extend SharpDetect, as well as demonstrated its capabilities on a real-world case study by using it to search for concurrency issues.

Within the introductory chapter, we set four major goals in connection with SharpDetect that we chose to fulfil in this project – extensibility, configurability, performance and the ability to run SharpDetect on multiple platforms. Regarding the extensibility, SharpDetect was used to implement two well-known algorithms – `Eraser` [7] and `FastTrack` [8] – that are used when searching for concurrency issues. Implemented configuration options offer powerful possibilities both for instrumenting and executing dynamic analysis. Then, in the fifth chapter, we measured the overhead of SharpDetect on selected subject programs and presented multiple solutions that were implemented mostly to reduce the overhead of both the instrumented programs and SharpDetect. Moreover, we tested that SharpDetect works on Windows and Linux, thus fulfilling also the goal about the ability to run on multiple platforms.

SharpDetect offers a simple console user-interface, which together with the plugin extensibility system can be used to integrate SharpDetect into various existing tools. For example, within an integrated development environment (IDE), SharpDetect could be utilized to directly mark statements by creating warnings on source code lines with potential issues. This and some other features and solutions to existing problems that were left as future work are covered in this thesis as well.

Despite the documented issues that were left for future work, we showed that SharpDetect is capable of performing dynamic analysis of .NET programs. We demonstrated its capabilities when using it to search for concurrency issues in a messaging library NetMQ [37]. The reports provided by SharpDetect generally allow for easy pinpointing the sources of issues because SharpDetect describes individual analysis events using available metadata. The reports can be even further enhanced to provide also a direct mapping from individual analysis events to the original source code lines as described in one of the propositions for future work in Section 4.4.5.

Bibliography

[1] Mark Dowson. The Ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997.

[2] Nancy G Leveson and Clark S Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[3] Mark Harman and Peter O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23. IEEE, 2018.

[4] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, number 6, pages 89–100. ACM, 2007.

[5] Cormac Flanagan and Stephen N Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2010.

[6] Announcement of .NET 5. `https://devblogs.microsoft.com/dotnet/introducing-net-5/` [Last access: 2019 October 07].

[7] 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):391–411, 1997.

[8] Cormac Flanagan and Stephen N Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI)*, number 6, pages 121–133. ACM, 2009.

[9] .NET Interpreter. `https://mattwarren.org/2017/03/30/The-.NET-IL-Interpreter/` [Last access: 2019 December 01].

[10] Common Language Infrastructure (CLI) the 3-rd edition (June 2005). `https://www.ecma-international.org/publications/files/ECMA-ST-WITHDRAWN/ECMA-335,%203rd%20edition,%20June%202005.pdf` [Last access: 2019 December 01].

[11] Roslyn Compiler Infrastructure. `https://github.com/dotnet/roslyn` [Last access: 2019 December 01].

[12] F# Compiler Infrastructure. `https://fsharp.github.io/FSharp.Compiler.Service/` [Last access: 2019 December 01].

[13] Mono.Cecil. https://github.com/jbevain/cecil
[Last access: 2019 December 01].

[14] Dnlib. https://github.com/0xd4d/dnlib
[Last access: 2019 December 01].

[15] Profiling (Unmanaged API Reference). https://docs.microsoft.com/en-us/dotnet/framework/unmanaged-api/profiling/
[Last access: 2019 December 01].

[16] Common Language Infrastructure (CLI) the 6-th edition (June 2012). https://www.ecma-international.org/publications/standards/Ecma-335.htm [Last access: 2019 December 01].

[17] The Book of the Runtime. https://github.com/dotnet/coreclr/tree/master/Documentation/botr
[Last access: 2019 October 07].

[18] Portable Executable (PE) Format Specification.
https://docs.microsoft.com/en-us/windows/win32/debug/pe-format
[Last access: 2019 December 01].

[19] Tail calls in F#.
https://devblogs.microsoft.com/fsharpteam/tail-calls-in-f/
[Last access: 2019 December 01].

[20] Subterranean IL: Compiling C# Exception Handlers.
https://www.red-gate.com/simple-talk/blogs/subterranean-il-compiling-c-exception-handlers/
[Last access: 2019 December 04].

[21] Using CrossGen to Create Native Images. https://github.com/dotnet/coreclr/blob/master/Documentation/building/crossgen.md
[Last access: 2019 December 01].

[22] Jump Stubs and JIT Compiler. https://github.com/dotnet/coreclr/blob/master/Documentation/design-docs/jump-stubs.md
[Last access: 2019 December 01].

[23] Command dotnet. https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet?tabs=netcore21 [Last access: 2019 November 04].

[24] Matt Pietrek. Metadata in. net-avoiding dll hell: Introducing application metadata in the microsoft. net framework. *MSDN Magazine*, pages 42–55, 2000.

[25] Bypass Signature Verification of Trusted Assemblies.
https://docs.microsoft.com/en-us/dotnet/standard/assembly/create-use-strong-named [Last access: 2019 December 01].

[26] GitLab CI/CD Documentation. https://docs.gitlab.com/ee/ci/
[Last access: 2019 December 01].

[27] NUnit. https://nunit.org/ [Last access: 2019 December 01].

[28] SharpLab. https://sharplab.io/ [Last access: 2019 December 01].

[29] DnSpy. https://github.com/0xd4d/dnSpy
[Last access: 2019 December 01].

[30] Erez Metula. *Managed Code Rootkits: Hooking into Runtime Environments*. Elsevier, 2010.

[31] Avoid DevPath – Bypassing GAC in .NET Framework. https://blogs.msdn.microsoft.com/suzcook/2003/08/15/avoid-devpath/
[Last access: 2019 December 01].

[32] PEVerify Tool. https://docs.microsoft.com/en-us/dotnet/framework/tools/peverify-exe-peverify-tool
[Last access: 2019 December 01].

[33] ILVerify Tool.
https://github.com/dotnet/corert/tree/master/src/ILVerify
[Last access: 2019 December 01].

[34] Program Database File Format.
https://github.com/Microsoft/microsoft-pdb
[Last access: 2019 December 13].

[35] Task Parallel Library (TPL). https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl
[Last access: 2019 December 12].

[36] Runtime Identifier (RID).
https://docs.microsoft.com/en-us/dotnet/core/rid-catalog
[Last access: 2019 December 12].

[37] NetMQ. https://netmq.readthedocs.io/en/latest/
[Last access: 2019 December 09].

[38] NetMQ Custom Clock Implementation. https://github.com/zeromq/netmq/blob/e4dfcf9e8190f85bf4fab9fc657e2c7da820c7f4/src/NetMQ/Core/Utils/Clock.cs#L88 [Last access: 2019 December 12].

[39] libzmq. https://github.com/zeromq/libzmq
[Last access: 2019 December 12].

List of Abbreviations

AOT ..... Ahead Of Time (used in context with compilation)

CIL ..... Common Intermediate Language

CLR ..... Common Language Runtime

CTS ..... Common Type System

GAC ..... Global Assembly Cache

GC ..... Garbage Collection

JIT ..... Just In Time (used in context with compilation)

PE ..... Portable Executable

RID ..... Runtime Identifier

TPL ..... Task Parallel Library

Attachments

Content of the attached CD

- The **SharpDetect** folder consists of the following subfolders:
  - **bin**: contains compiled SharpDetect assemblies, except for plugins.
  - **Examples**: contains a copy of the folder src/SharpDetect.Examples
  - **Plugins**: contains compiled SharpDetect analysis plugins.
- The **src** folder contains source code of the whole SharpDetect solution:
  - **SharpDetect.**{**Common, Console, Core, Injector, Plugins**} folders contain the main SharpDetect modules.
  - **SharpDetect.**{**FunctionalTests, UnitTests**} folders contain the testing modules of SharpDetect.
  - **SharpDetect.Examples** contains subject programs together with the configuration as presented in the thesis
  - **.gitlab-ci.yml** is a continuous integration pipeline for SharpDetect
  - **setup.ps1** is a powershell script that can be used to generate the content of the **bin** folder.
- The **README.txt** file contains information about the content of the attached CD and author's email address.