DOI: 10.15514/ISPRAS-2024-36(3)-4

Support of Visual Basic .NET in SharpChecker static analyzer

[1,2] *V.S. Karcev*, ORCID: 0000-0001-7482-0835 <*karcev.vs@ispras.ru*>
[1,3] *V.N. Ignatyev*, ORCID: 0000-0003-3192-1390 <*valery.ignatyev@ispras.ru*>

[1] *Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2]*Moscow Institute of Physics and Technology,*
9, Institutsky lane, Dolgoprudny, 141701, Russia.
[3] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. This paper presents the implementation of static analysis for Visual Basic .NET (VB.NET) within the industrial tool SharpChecker. Leveraging the Roslyn compiler framework, VB.NET analysis was integrated into SharpChecker, enabling static code analysis for VB.NET projects. The process involved building support for VB.NET projects, creating a comprehensive test suite, implementing a source code indexer, and adapting existing analyzers to support VB.NET syntax nodes and operations. Evaluation of translated tests and real-world projects demonstrated production-acceptable analysis quality, paving the way for improved maintenance of VB.NET projects. Additionally, the study highlighted SharpChecker's capability for cross-language analysis, showcasing its ability to handle mixed C# and VB.NET projects efficiently.

Keywords: static code analysis; vulnerabilities detection; VB.NET.

For citation: Karcev V.S., Ignatyev V.N. Support of Visual Basic .NET in SharpChecker static analyzer. Trudy ISP RAN/Proc. ISP RAS, vol. 36, issue 3, 2024. pp. 49-62. DOI: 10.15514/ISPRAS-2024-36(3)-4.

Поддержка Visual Basic .NET в статическом анализаторе SharpChecker

[1,2] *В.С. Карцев,* ORCID: 0000-0001-7482-0835 <*karcev.vs@ispras.ru*>
[1,3] *В.Н. Игнатьев,* ORCID: 0000-0003-3192-1390 <*valery.ignatyev@ispras.ru*>

[1] *Институт системного программирования РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский Физико-Технический Институт,
Россия, 141701, Долгопрудный, Институтский переулок, д. 9*
[3] *Московский государственный университет имени М.В. Ломоносова,
Россия, 119991 Москва, Ленинские горы, д. 1.*

Аннотация. В этой статье представлена реализация статического анализа для Visual Basic .NET (VB.NET) в промышленном инструменте SharpChecker. Используя фреймворк компилятора Roslyn, анализ VB.NET был интегрирован в SharpChecker, что позволило выполнять статический анализ кода для проектов VB.NET. Процесс включал в себя создание поддержки для проектов VB.NET, создание всеобъемлющего набора тестов, реализацию индексатора исходного кода и адаптацию существующих анализаторов для поддержки узлов и операций синтаксиса VB.NET. Оценка переведенных тестов и реальных проектов продемонстрировала приемлемое для производства качество анализа, проложив путь для улучшенного обслуживания проектов VB.NET. Кроме того, исследование подчеркнуло возможности SharpChecker для кросс-языкового анализа, продемонстрировав его способность эффективно обрабатывать смешанные проекты C# и VB.NET.

Ключевые слова: статический анализ кода; обнаружение уязвимостей; VB.NET.

Для цитирования: Карцев В.С., Игнатьев В.Н. Поддержка Visual Basic .NET в статическом анализаторе SharpChecker. Труды ИСП РАН, том 36, вып. 3, 2024 г., стр. 49–62 (на английском языке). DOI: 10.15514/ISPRAS–2024–36(3)–4.

1. Introduction

Visual Basic .NET (hereinafter referred to as VB.NET) is an interpreted, object-oriented language with static typing, developed by Microsoft Corporation. VB.NET was inspired by Visual Basic 6.0. It provides human-friendly syntax like its predecessor. However, it differs significantly from its predecessor. Microsoft explains that it can help people who don't know programming to understand and discuss tasks with others. The most noticeable difference from its predecessor is true object orientation. Also, there are much more differences between these languages, so VB.NET is more similar to other .NET languages like C#.

This enables the implementation of static analysis for VB.NET in existing solutions — in SharpChecker [1]. SharpChecker is an industrial static analyzer for C#. It uses Roslyn [2] to compile code, build an abstract syntax tree, make symbols table, etc. Roslyn supports C# and VB.NET. So, implementing of VB.NET static analysis in SharpChecker must be simpler than developing a new analyzer. Also, there are cross-language projects written in C# and VB.NET (for example, Roslyn itself).

The work of SharpChecker can be divided into several stages. First of all, SharpChecker, using the Roslyn [2] infrastructure, intercepts the compilation and assembles the solution. Next, the analysis phase starts, consisting mainly of analysis of the abstract syntax tree and symbolic execution. Next, scalable interprocedural analysis is performed, sensitive to control flow [3]. Symbolic execution in SharpChecker allows for false positives and missed errors. Its goal is to find the maximum number of errors in the minimum time for a given percentage of true positives. At this stage, the control flow graph, possible values of variables, and feasibility of conditions are analyzed. Using symbolic execution, you can find errors related to, for example, unreachable code or the use of a disposed resource. After this, the labeled data is analyzed, with the help of which it is possible to detect vulnerabilities related, for example, to data security. In addition, SharpChecker searches and stores all references to symbols, such as types, variables, methods, etc. This allows for source code navigation in the user interface.

1.1 Relevance

VB.NET is the tenth most popular language according to TIOBE [4] ranking. It is used in many companies to maintain old projects. Mostly, such projects don’t have a detailed documentation and may contain many hidden errors.

So, static analysis can help to fix and maintain legacy VB.NET projects. It can also ease the transition from Visual Basic 6.0 to VB.NET by finding and reporting code problems to programmers.

1.2 Motivational example

VB.NET may contain many types of errors. Here is DEREF_AFTER_NULL error found in Roslyn compiler.

```
 1  Dim initializerOpt = node.InitializerOpt
 2  If initializerOpt Is Nothing OrElse ... Then
 3    If node.Bounds.Length = 1 Then
 4      Dim lastIndex = node.Bounds(0)
 5      If ... Then
 6        ...
 7        ReportDiagnostic(diag, initializerOpt.Syntax, ...)
 8        _hasErrors = True
 9      End If
10    End If
11  End If
```

Listing 1. Declaration of IgnoreAccessibility property

In this example, first of all, the value of the `initializerOpt` variable is compared with `Nothing` in line 2. If this condition is met (that is, if the value of the `initializerOpt` variable is `Nothing`), a function `ReportDiagnostic` can be called. In parameters of this function variable `initializerOpt` is dereferenced. Thus, under certain conditions, `Nothing` dereferencing is possible in this case.

2. Related works

VB.NET is not supported by the majority of industrial static analyzers [5]. For example — Klocwork [6] and Coverity [7] have no VB.NET support, although these analyzers support C#.

Despite this, VB.NET is supported in some static analyzers such as ReSharper [8-9], SonarQube CE [10-11] and Kiuwan [12].

2.1 ReSharper

Many static analyzers, such as ReSharper, are designed primarily for quickly analyzing code to assist the developer immediately while writing code. This leads to a significant reduction in the accuracy of such tools, which allows them to find only the simplest special cases. So, ReSharper does not take into account many factors that require resource-intensive analysis in advance. For this reason, such tools are also unable to detect errors that can only be found with complex analysis such as symbolic execution or analysis of tainted data.

2.2 SonarQube CE

SonarQube CE is used to detect errors, vulnerabilities, and code smells based on rules. However, most rules for VB.NET are designed to detect code smells. Most of the rules in the bugs and vulnerabilities categories are aimed at finding simple cases, such as finding recursive inheritances. However, there are also more complex rules — e. g. no lock release on one or more of the execution paths [13].

2.3 Kiuwan

This product focuses primarily on code security issues such as buffer overflows, command injections, cross-site scripting, and SQL injections. At the same time, these types of vulnerabilities are only a small part of the problems detected by the SharpChecker tool.

There are few analyzers that support the VB.NET language, and the existing ones often implement only a small part of the required functionality. Therefore, the implementation of the VB.NET analysis within the SharpChecker tool is important.

3. Problem statement

VB.NET is based on the same intermediate language CIL [14] as C# and is compiled by the same compiler — Roslyn. So, this makes it possible to reuse existing SharpChecker infrastructure for VB.NET program analysis. So, the main goal of this work is to implement VB.NET static analysis within the industrial tool SharpChecker.

Implementation of new .NET language in SharpChecker can be divided into several tasks:

- Create representative test-set to understand problems and incompatibilities;
- Implement source code navigation to analyze warnings in real projects;
- Add new VB.NET specific processing in analyzers:
  - abstract syntax tree analyzers;
  - symbolic execution analyzers;
  - taint analyzers.

In general, Roslyn has separate modules for building an abstract syntax tree for each .NET language. But it also has modules for code analysis unification. For example, Roslyn has IOperations that can unify the analysis of identical features for all supported languages.

4. The Approach

4.1 Test Base

To speed up the development, a decision was made to start with creating a complete testbase with maximum possible coverage. SharpChecker has many separate analyzers for different error types. Also, it has several analyzers to collect information about the code.

It is necessary to measure the accuracy of the analysis results. This leads to the necessity of a large test base. Moreover, tests can help to debug incorrect behavior. In order to cover all analyzers and error types to check analyzers' behavior in precise, it was decided to create an automatic test translator from C# to VB.NET.

Test in SharpChecker is a syntactically correct compiled program, which may contain intentional errors. Also, in the source code of the test, there is a marking of errors made in the form of comments with information about the error. During the test execution, this source code is passed into the SharpChecker, after which its results are compared with the markup in the test. The test is considered passed if the SharpChecker detects all errors made, doesn't detect non-existent errors, and doesn't terminate abnormally.

At first, the translator traverses the SharpChecker's sources to find and parse all files with tests. It creates an abstract syntax tree of every file using Roslyn and finds all calls of verification methods. Such methods contain the source code of the test as an argument, passed as a multiline string. All these strings are converted to VB.NET by a separate translator's subsystem. After translating the initial test is replaced with the result of translation in the initial syntax tree. When all tests are translated, the whole syntax tree is exported into a new file.

4.1.1 Symbols renaming

The main problem with test translation from C# to VB.NET at all is case-insensitivity in VB.NET. However, case-insensitivity is observed only within a single compilation unit. Accessing some symbols from another project it is necessary to use the same case as this symbol was originally defined. This problem led to the necessity of symbol renaming. All symbols that have differences only in case must be stored and enumerated. After that, all symbols defined in the scope of the compilation unit must be renamed according to enumeration. Other symbols, e. g. those that are taken from external libraries, are not renamed.

For example, let's rename such symbols in code on listing 2.

```
using System;
public namespace TEST {
  public class Test {
    public int console = 0;
    public void test() {
      Console.WriteLine($"console = {console}");
    }
  }
}
```

Listing 2. Example of code with case-sensitive symbols

In this example, names `TEST`, `Test`, and `test` are indistinguishable, as `Console` and `console` too. The symbols will be renamed as follows: `TEST1`, `Test2`, `test3`, `Console1`, and `console2`. However, since the symbol `Console` was declared in an external library, renaming it could lead to compilation errors and, as a result, it won't be translated. After renaming example will look like on listing 3.

```
using System;
public namespace TEST1 {
  public class Test2 {
    public int console2 = 0;
    public void test3() {
      Console.WriteLine($"console = {console2}");
    }
  }
}
```

Listing 3. Example with renamed symbols

4.1.2 Test translating

In order to translate source code from C# to VB.NET translator builds an abstract syntax tree of the source. After that, it generates a new abstract syntax tree with corresponding VB.NET syntax nodes for every C# syntax node. Some expressions must be completely replaced with others. For example, in VB.NET variables of reference type must be compared with `Nothing` only with `Is` or `IsNot` operators instead of `==` and `!=`. Additionally, comparing `char` with `int` requires explicit conversion to the same type.

Translation result for example above is shown on listing 4.

```
Namespace TEST1
  Public Class Test2
    Public console2 As Integer = 0
    Public Sub test3()
      Console.WriteLine($"console = {console2}")
    End Sub
  End Class
End Namespace
```

Listing 4. Translated example

4.1.3 Translation results

SharpChecker contains 2680 tests, written in C#. 1928 of them were successfully translated to VB.NET. All tests that could not be translated contain features not supported in VB.NET, making conversion impossible. For example, VB.NET doesn't have `dynamic` type.

Automatic test translation enables the development of a representative test base for VB.NET from scratch. This test base is almost equivalent to the one for C#, which has been developing for over 5 years. It contains tests for each of the FIXME types of error, supported by SharpChecker.

4.2 Implementing source code navigation

It is necessary to implement navigation for VB.NET. A source code indexer is used to collect code data. It is a component that collects information about symbols. It is used to support navigation (quick jumps to declarations, definitions and usages of symbols such as variables, functions, or classes) through the source code in a GUI. The necessity of indexer is significant because it helps to analyze warnings in huge projects and make decisions about analysis quality. So, to implement an indexer for VB.NET it is necessary to find all definitions, declarations, and usages of every single symbol of source code.

It is impossible to use the existing indexer for C# because of differences in syntax between C# and VB.NET. For instance, VB.NET has integrated XML syntax, allowing for the inclusion of variables within XML elements.

As part of the implementation of VB.NET support, a syntax indexer was implemented. The implementation is basically similar to the implementation for C#, however, some syntax patterns specific to VB.NET were taken into account (such as the usage of variables within the XML syntax). As a result, the syntax indexer successfully parses projects and enables code navigation in the GUI.

4.3 Major incompatibility reasons

Most part of the inaccuracies in SharpChecker are related to the fact that VB.NET and C# have different types of syntax nodes in an abstract syntax tree. As a result, analyzers directly reliant on the analysis of abstract syntax trees may produce incorrect results. Usage of syntax nodes in any context in SharpChecker may lead to incorrect results in VB.NET analysis.

There are also several analyzers that provide their results to other analyzers. Usage of an abstract syntax tree in such analyzers can lead to completely incorrect results.

4.4 Abstract syntax tree analyzers

Abstract syntax tree analyzers (hereinafter AST analyzers), as the name suggests, provide analysis based only on an abstract syntax tree (or other variants of source code representation, e.g., operations tree or symbol table). The main features of these analyzers include high analysis speed, low resource consumption, and independence from other SharpChecker components. It leads to the need to adapt each single AST analyzer separately to let it analyze VB.NET code.

As it was mentioned above Roslyn has some unification mechanisms as IOperations and Symbols are. Some analyzers that work only with such abstractions can perform VB.NET analysis as they are. But other AST analyzers must be rewritten to support C# and VB.NET syntax nodes both or to work with IOpeations and Symbols only. There are 61 AST analyzers in SharpChecker and most of them working with C# syntax nodes, but some of them are already working with IOperations and Symbols only. For example, the UselessCall analyzer is already using IOperations to perform analysis. Also, there are some analyzers that don't need syntax or operation tree at all e.g., DuplicateEnumMember analyzer, that performs analysis only with symbol table.

To prepare AST analyzer to work with VB.NET it is necessary to follow these steps:

- replace all syntax nodes that have operation analogs with operations (sometimes it requires rewriting analysis logic for such nodes);
- add VB.NET syntax nodes processing as it is implemented for C#;
- find all hard coded names or constructions, that are incompatible with VB.NET (for example VB.NET has keyword `Nothing` instead of `null` in C#);
- check if the analyzer started working correctly with VB.NET.

Since AST analyzers are independent of other analyzers and their modification follows the same scheme, at the moment only a part of them was modified to check their functionality with VB.NET. These analyzers are:

- `EmptyInterface` — added processing of VB.NET syntax nodes along with C# nodes;
- `FloatingPointEquality` — rewritten with IOperations;
- `RealIntegerComparison` — rewritten with IOperations;
- `ShadowedName` — syntax nodes analysis was replaced with analysis of IOperations and Symbols.

Testing of the listed analyzers after modification showed that they began to work correctly with the code on VB.NET.

4.5 Symbolic execution analyzers

The symbolic execution stage performs scalable interprocedural path-sensitive analysis [3]. At this stage, false positives and missed errors are tolerated. The main goal of this type of analysis is to search for the maximum number of errors in a minimum time with a fixed ratio of true and false positives. At this stage, the control flow graph constructed in the syntax tree analysis phase is analyzed, the variables are parameterized and the transition conditions for the control flow graph are preliminarily calculated. Using this method, complex errors associated with scenarios such as unreachable code [15] or the use of a disposed resource [16] can be identified.

Symbolic execution analyzers are more complex than AST ones. Also, such analyzers mostly have a complex dependency graph.

The dependency graph of the UnreachableCode analyzer is depicted in figure 1 as an example. It can be seen that the results of this analyzer are highly dependent on the correct operation of many other analyzers.

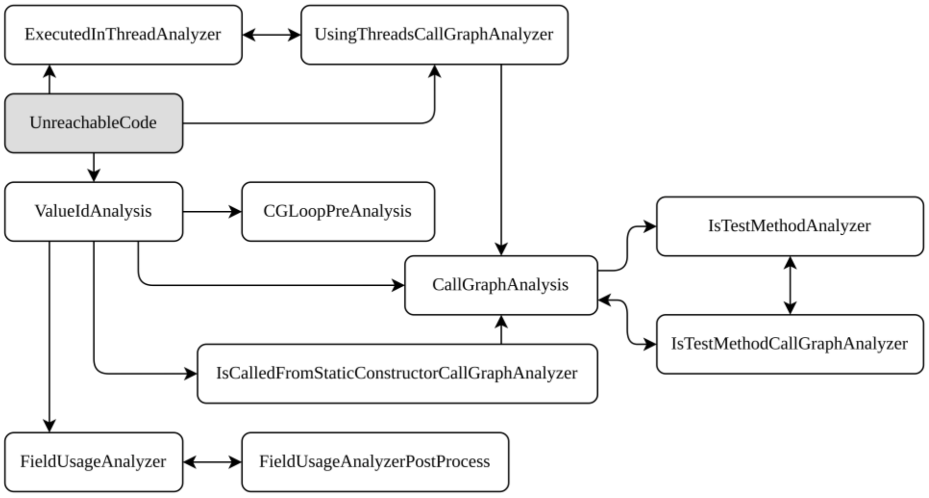
ExecutedInThreadAnalyzer
UsingThreadsCallGraphAnalyzer
UnreachableCode
ValueIdAnalysis
CGLoopPreAnalysis
IsTestMethodAnalyzer
CallGraphAnalysis
IsTestMethodCallGraphAnalyzer
IsCalledFromStaticConstructorCallGraphAnalyzer
FieldUsageAnalyzer
FieldUsageAnalyzerPostProcess

Fig. 1. UnreachableCode dependencies

Some symbolic analyzers collect some information and provide it to others. These analyzers make up the symbolic engine. The most important part of preparing symbolic analyzers to work with VB.NET code is to adapt the engine.

4.5.1 Symbolic engine

Symbolic engine collects a lot of useful information for further analysis. It analyzes paths, contexts, variable values, conditions, and much more. In order to make symbolic analyzers work correctly with VB.NET it is important to prepare the whole engine because one minor inaccuracy in the engine may lead to the absolutely incorrect results in all symbolic analyzers.

In general, the symbolic engine has the same issue with the analysis of syntax nodes. But basically, this analysis cannot be replaced by the analysis of IOperations or Symbols, so it is necessary to duplicate the processing for VB.NET syntax nodes. Also, some minor differences were found:

- symbols that represent properties in C# has property `UnderlyingSymbol` that contains information about backing field, when such symbols from VB.NET code contain backing field directly as field of initial symbol and have no `UnderlyingSymbol`;
- method symbols in VB.NET have no property to indicate that method is partial and it has no implementation — it is necessary to use indirect signs;
- `Nothing` in VB.NET has no constant value unlike `null` in C#.

4.5.2 Symbolic analyzers

After the implementation of all the improvements in the symbolic engine, the symbolic analyzers should generally work correctly. However, some analyzers use syntax nodes as well as a symbolic engine. For example, UnreachableCode or NullDereference analyzers use syntax nodes to clarify warning message or add tags. Also, sometimes syntax nodes can be used to detect some corner cases or to find additional information.

4.6 Taint analyzers

Taint analysis is based on propagating tainted data. It helps to find security issues in source code. In SharpChecker taint analysis depends on the CallGraphAnalysis analyzer, which is used by the symbolic engine, so it has been already fixed in the previous stage. It leads to the necessity of adding

VB.NET support in CallGraphAnalysis only. It has already been done on the stage of the symbolic engine.

So, taint analysis showed good analysis quality without any additional modifications.

5. Evaluation

The quality of the analysis was measured on translated tests and real projects. After implementing the analysis of the VB.NET syntax node and IOperations in most of the symbolic engine analyzers, SharpChecker began detecting errors in the VB.NET source code.

5.1 Tests results

It could be assumed that generated tests cover the most error types. Analyzers working on the AST level were not fully rewritten, so let's consider other analyzers.

Table 1. Tests evaluation

	Symbolic Execution	Taint	Dataflow
Pass	1177	77	39
Fail	179	13	11
Pass rate	86.8%	85.5%	78.0%

Table 1 shows the results of running tests automatically generated for VB.NET. The table shows the number of passed and failed tests for each of the analyzed types of analyzers. The table also shows the percentage of tests passed for each type of analysis. Among the failed tests, both missed errors and false positives were found.

Based on these results, it can be concluded that as a result of the changes, SharpChecker received VB.NET support, but some corner cases were left uncovered.

5.2 Real projects warnings

As a result of testing on real projects, it turned out that SharpChecker is able to detect errors on real projects after adding support for VB.NET. Several examples of found errors are given below.

5.2.1 USELESS_CALL

Warning on listings 5, 6 was found in Roslyn in file `TupleMethodSymbol.vb` in line 131.

The USELESS_CALL warning was found in the code shown in the listing. Indeed, the function `MergeInfo` called in line 3 has no side effects, and its return value is not assigned anywhere.

5.2.2 DEREF_AFTER_NULL

Warning on listing 7 was found in Roslyn in the file `Binder_Lookup.vb` in line 1909.

In VB.NET there are differences between the operators `And` and `AndAlso`. The regular `And` operator evaluates both sides of an expression, even if the final value can be calculated from only the first operand. In this example, the developer assumed that if the `container` variable is equal to `Nothing` (which is checked by the first operand), then the second operand will not be calculated and dereferencing will not occur. However, due to the specifics of the language, in this case, both operands will always be evaluated, which leads to the possibility of a `Nothing` dereferencing error.

5.2.3 HANDLE_LEAK

Warning on listing 8 was found in Roslyn in the file `Program.vb` in line 75.

```
Friend Overrides Function Info() As Info
  Dim useSiteDiagnostic As Info = MyBase.Info()
  MyBase.MergeInfo(useSiteDiagnostic,
    Me._underlyingMethod.GetUseSiteErrorInfo())
  Return useSiteDiagnostic
End Function
```

Listing 5. USELESS_CALL warning

```
Function MergeInfo(
  first As Info, second As Info) As Info
  If first Is Nothing Then
    Return second
  End If
  If second Is Nothing OrElse
    second.Code <> HighestPriorityError Then
    Return first
  End If
  Return second
End Function
```

Listing 6. Definition of `MergeInfo`

```
If container IsNot Nothing And
  container.SpecialType = SpecialType.System_Void Then
  Return
End If
```

Listing 7. DEREF_AFTER_NULL warning

```
1  Function GetChecksum(filePath As String) As String
2    Dim fileBytes = File.ReadAllBytes(filePath)
3    Dim func = SHA256.Create()
4    Dim hashBytes = func.ComputeHash(fileBytes)
5    Dim data = BitConverter.ToString(hashBytes)
6    Return data.Replace("-", "")
7  End Function
```

Listing 8. HANDLE_LEAK warning

As can be seen, the resource `func` created in line 3 is not disposed until the end of the function, after which it goes out of scope. So, this is indeed a mistake.

However, it is possible to observe a fairly large number of false positive warnings at the symbolic level analyzers. Such warnings will be eliminated in the future by debugging and finding incorrect processing of the source code on VB.NET.

Thus, as a result of the changes, SharpChecker received support for the VB.NET analysis. As a result of the analysis of the Roslyn project, warnings were found in the VB.NET source code. Some of the found warnings turned out to be true positives.

5.3 Real projects quality

To determine the quality of the analysis on real projects, Roslyn was analyzed. 50 warnings were marked for each considered analyzer. The results are shown in table 2.

Table 2. Real projects evaluation

Warning type	Warning's count		TP rate	
	VB.NET	C#	VB.NET	C#
UNUSED_VALUE	325	694	90.0%	99.6%
DEREF_AFTER_NULL	57	54	38.6%	48.4%
UNREACHABLE_CODE	331	428	37.0%	58.3%

The table lists the number of warnings found and the percentage of the ratio of the number of true positives to the total number of marked ones for each of the considered analyzers. The marking of warnings was done manually.

The results show that the analyzers detect warnings in VB.NET source code with an accuracy comparable to that for C#. In some cases, the accuracy is lower (for example, in DEREF_AFTER_NULL and UNREACHABLE_CODE), it is explained by minor differences in the structure of languages and their representation in Roslyn.

5.4 Cross-language analysis

While testing VB.NET support on real projects, it was revealed that SharpChecker is able to analyze cross-language projects. As it turned out from test runs of Roslyn, SharpChecker can analyze cross-language projects without quality losses in the appropriate time. Both errors in VB.NET and errors in C# were simultaneously detected. In addition, it is worth noting that discovered errors were simultaneously associated with code in both languages. Thus, the trace of such errors was found both in the C# code and in the VB.NET code.

Before support for VB.NET analysis, analysis lasted 30 minutes and after implementing it time has increased to 44 minutes. It can be explained by the increased number of sources to analyze. Roslyn contains 2.25 million lines of C# code and 1.52 million of VB.NET code. So, increasing analysis execution time correlates with increasing code size. It is also worth noting that the results of the analysis of the source code in C# have not changed.

5.5 Comparison with SonarQube

To compare the quality of the analysis, the Roslyn project was analyzed using the SonarQube CE tool. 76 errors and more than 4 thousand code smells were found. Detected errors include the following types of errors:

- incorrect getter name;
- identical right and left sides of a logical expression;
- implicit casting to an integer type in a bitwise shift;
- identical blocks in a branching;
- identical conditions in a branching.

These errors are quite simple and can be detected without the use of in-depth analysis. At the same time, more complex errors, such as memory leaks and unreachable code, were not detected by the SonarQube CE tool.

6. Conclusion

SharpChecker is ready to analyze VB.NET source code without major changes. There are many minor changes to be done in SharpChecker for complete support of VB.NET, but now it works with acceptable quality. However, it is necessary to overcome a number of problems in the detectors that arise due to differences in languages and differences in the implementation of methods and abstractions for their analysis in Roslyn. This requires detailed debugging of failed tests and the addition of edge case processing in all analyzers included in SharpChecker.

References

[1]. V. Koshelev, V. Ignatiev, A. Borzilov, and A. Belevantsev. SharpChecker: static analysis tool for C# programs. Programming and Computer Software, 43(4):268–276, 2017.

[2]. dotnet/roslyn: The Roslyn .NET compiler provides C# and Visual Basic languages with rich code analysis APIs.https://github.com/dotnet/roslyn. [Online, accessed 23.10.2021].

[3]. R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. ACM Comput. Surv., 51(3), 2018. DOI: 10.1145/3182657. URL: https://doi.org/10.1145/3182657.

[4]. TIOBE Index for ranking the popularity of Programming languages. https://www.tiobe.com/tiobe-index, 2022.

[5]. Wikipedia contributors. List of tools for static code analysis — Wikipedia, the free encyclopedia, 2024. URL: https://en.wikipedia.org/w/index.php?title=List_of_tools_for_static_code_analysis&oldid=1218561224. [Online; accessed 15-April-2024].

[6]. W. Wei, M. Yunxiu, H. Lilong, and B. He. From source code analysis to static software testing. In 2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA), pages 1280–1283. IEEE, 2014.

[7]. A. Almossawi, K. Lim, and T. Sinha. Analysis tool evaluation: coverity prevent. Pittsburgh, PA: Carnegie Mellon University:7–11, 2006.

[8]. E. Firouzi and A. Sami. Visual studio automated refactoring tool should improve development time, but resharper led to more solution-build failures. In 2019 IEEE Workshop on Mining and Analyzing Interaction Histories (MAINT), pages 2–6. IEEE, 2019.

[9]. Resharper features. https://www.jetbrains.com/ru-ru/resharper/features/, 2022.

[10]. V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi. Are sonarqube rules inducing bugs? In 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 501–511. IEEE, 2020.

[11]. G. A. Campbell and P. P. Papapetrou. SonarQube in action. Manning Publications Co., 2013.

[12]. Common vulnerabilities. https://www.kiuwan.com/common-vulnerabilities/, 2024.

[13]. Vb.net static code analysis. https://rules.sonarsource.com/vbnet/, 2024.

[14]. Wikipedia contributors. Common intermediate language — Wikipedia, the free encyclopedia, 2024. URL: https://en.wikipedia.org/w/index.php?title=Common_Intermediate_Language&oldid=1218588686. [Online; accessed 16-April-2024].

[15]. V. N. Ignatiev, V. K. Koshelev, A. I. Borzilov, A. A. Belevantsev, N. V. Shimchik, and M. V. Belyaev. Detector of unreachable code in C# programs of the static analysis tool “SharpChecker”, 2017.

[16]. U. V. Tyazhkorob, V. N. Ignatiev, and A. A. Belevantsev. Finding uses of a disposed resource in source code in C# using static analysis methods. Proceedings of the Institute of System Programming RAS, 34(6):41–50, 2022.

Информация об авторах / Information about authors

Вадим Сергеевич КАРЦЕВ – студент магистратуры Физтех-школы Радиотехники и Компьютерных Технологий МФТИ, сотрудник ИСП РАН. Научные интересы: компиляторные технологии, статический анализ программ, статическое символьное выполнение, поиск дефектов в исходном коде.

Vadim Sergeevitch KARCEV is a master student at the Department of Radio Engineering and Computer Technologies of MIPT, an employee of the ISP RAS. Research interests: compiler technologies, static program analysis, static symbolic execution, defect search in source.

Валерий Николаевич ИГНАТЬЕВ – кандидат физико-математических наук, старший научный сотрудник ИСП РАН, доцент кафедры системного программирования факультета ВМК МГУ. Научные интересы включают методы поиска ошибок в исходном коде ПО на основе статического анализа.

Valery Nikolayevich IGNATYEV – Cand. Sci (Phys.-Math.), senior researcher at Ivannikov Institute for System Programming RAS and associate professor at system programming division of CMC faculty of Lomonosov Moscow State University. He is interested in techniques of errors and vulnerabilities detection in program source code using static analysis.