DECOMPILING JAVA BYTECODE: FROM COMMON JAVA TO NEWEST JDK8 FEATURES, FROM OBSOLETE DECOMPILERS TO CUTTING EDGE TECHNOLOGY

Jozef KOSTELANSKY, Lubomir DEDERA

Abstract: Countless various malware families provide huge variety of functionalities which allow them to do many malicious activities. These conditions led to the development of many different analysis methods. In this paper, we focused on reverse engineering, which is elementary part of static analysis. We evaluate current Java bytecode decompilers. We evaluate the output from current Java bytecode decompilers in this paper using test samples and metrics from previous surveys in 2003 and in 2009. Quality boost is really significant between actual results and the results based on research from 2009, in contrast with only slight improvement between researches in 2003 and 2009. Even though we were witnesses of rapid quality boost, still any of the decompilers pass all the tests acceptably. We also give reasons why outdated decompilers from previous research perform almost in the same way.

Keywords: java, reverse engineering, bytecode, decompilation.

1 INTRODUCTION

There are more definitions of malware. Generally, malware (malicious software) is a program focused on collecting data (key logger), disrupt regular operations (ransomware) or to abuse infected systems in favor of the attacker (bitcoin miners, botnets) [1].

In general, there are two methodologies to dissect malware behavior: static and dynamic analysis. Static analysis comprises of all approaches to analyze sample without the sample being executed. On the other hand, in dynamic analysis, the sample is executed, and its behavior is observed [2]. Malware creators are daily trying to circumvent the above-mentioned analysis methods; this usually means usage of obfuscation and encryption, to circumvent with static analysis and usage of some sandbox detecting and antidebugging techniques to evade dynamic analysis.

Based on the stats provided by statcounter.com, Android has the biggest market share, not even from smartphones, but in general. This, together with valuable data which users holds in their smartphones, makes Android very frequent target for malware campaigns. This situation was predicted in 2009 [3]. Since most Android applications are developed in Java, each of the mentioned methodologies has its pros and cons. Since by means of static analysis it is possible to check all malware execution paths, our primary attention will be concentrated on static analysis and obviously we have focused on Java decompilation.

To accomplish complex static analysis, it is ideal to have original source code, but since we usually don't have access to malware source code, we need to make it in our own way. Although it is easier to decompile Java bytecode than machine code, there are still cases that the decompilers have problems to cope with. These include variable casting or exception handling. Evaluation is focused on those problem areas.

There are many Java decompilers available, some of them are paid, some of them free and open sourced. We have stated two main goals of this research:

- To evaluate current Java bytecode decompilers;
- To compare previous results with our actual results.

In this paper, we have followed the ideas presented in [38] and tried to extend them. The idea has been presented for the first time at the conference [38] and now, based on the conclusion from that work, we tried to elaborate the idea presented there; namely, we will look closer, how current decompilers are able to cope with the newest JDK features like lambdas, string switches and others.

2 JAVA DECOMPILERS EVALUATION STRATEGY

The output of a Java decompiler can, crudely, be divided into three categories:

1. Semantically and syntactically correct;
2. Syntactically correct and semantically incorrect and
3. Syntactically incorrect.

Since last evaluations [4] [5] of Java decompilers were published several years ago and since then most of them became outdated, obsolete and, on the other hand, some new decompilers have been developed. We have used the same samples for decompilation and also the same metrics for evaluation of decompilers output, because we want to compare how they have changed in time. In addition, we recompiled samples with current javac (version 1.8.0_131), current jasmin (version 2.4) [6] and we have tested several new decompilers in our research (cfr [7], fernflower [8], Krakatau [9], Procyon [10]).

The following table contains summary of the measured metrics.

Table 1 Decompilation correctness classification [11]

Score	Semantics	Syntax	Output result	Examples
0	correct	correct	semantically and syntactically correct program with perfect/good source code layout	perfect decompilation
1	correct	correct	semantically and syntactically correct program with `ugly' source code layout and/or no type inference	unreconstructed control ow statements, unreconstructed string concatenation, unused labels, no type inference
2	incorrect	incorrect	easy to correct syntax errors which produce a semantically correct program	boolean typed as int, missing variable declaration
3	incorrect	incorrect	difficult (but possible) to correct syntax errors which produce a semantically correct program	code with goto statements
4	incorrect	incorrect	very difficult (or nearly impossible) to correct syntax errors required to produce a semantically correct program	invalid variable use, obviously incorrect code, massive source re-write required
5	incorrect	correct	easy to correct semantic errors which produce a semantically correct program	missing typecasts
6	incorrect	correct	difficult (but possible) to correct semantic errors which produce a semantically correct program	incorrect control ow
7	incorrect	correct	very difficult (or nearly impossible) to correct semantic errors required to produce a semantically correct program	incorrectly nested try-catch blocks, massive source re-write required
8	incorrect	incorrect	incomplete decompilation	missing large sections of source, missing inner classes
9	Fail	Fail	decompiler fails upon execution/produces no source output	decompiler fails to parse arbitrary bytecode

Table 1 contains a summary of decompilation correctness classification published in [11]. For each input sample and the corresponding decompiled pair, a score between 0 and 9 was given. Referring to the table I., score 0 means perfect decompilation and score 9 means that no output decompiled code was produced.

3 EVALUATION OF CURRENT JAVA BYTECODE DECOMPILERS

The following decompilers were tested. First, here are the decompilers which were used in the previous survey [4].

Mocha [12] was developed in 1996 by Dutch developer Hanpeter van Vliet, alongside an obfuscator named Crema. It is used here just for historical reason, since it is not useful today, because it does not support bytecode generated with the current javac.

SourceTec [13] (Sothink Java Decompiler) supports analyzing Java class files and generating equivalent and compliable Java source codes. This unmaintained tool is a patch to Mocha and also does not support bytecode generated with the current javac.

SourceAgain [14] was another commercial decompiler, which is now unmaintained and unavailable to download, but a web version is available.

Jad [11] (Java Decompiler) is, as of August 2011, an unmaintained decompiler for the Java programming language. Jad provides a command-line user interface to extract source code from class files. The official site http://www.kpdus.com is no longer accessible, but it is possible to download it from many mirrors. It was one of the best decompilers, but it was not open source and was developed in C++, so it cannot be easily reverse engineered. Probably, this is the most popular Java decompiler, but primarily of this age only. Written in C++, so it is very fast.

JODE [15] is an open-source decompiler and obfuscator. The latest version 1.1.2-pre1 was released February 24, 2004. It thus doesn't handle many language constructs that were introduced after JDK 1.3.

jReversePro [16] is another open-source disassembler and decompiler project which is currently unmaintained.

Dava [17] [18] is part of Soot Java optimization framework which is still under development and actually provides nightly builds. Soot framework is developed by Sable Research Group at McGill University in Montreal, Quebec, Canada.

Jdec [19] is another abandoned open source java decompiler with support up to Java 1.4.

Java Decompiler [20] aims to develop tools in order to decompile and analyze Java 5 byte code and the later versions.

Following decompilers are relatively new and, except Procyon, are currently under development.

Cfr [7] is a free, but not open sourced Java decompiler which support modern Java features like Java 8 lambdas. Last version 0_122 was released on 2017-06-12.

Fernflower [8] is a promising analytical Java decompiler, now becomes an integral part of IntelliJ 14 [21].

Krakatau [9] [22] currently contains three tools – a decompiler and disassembler for Java class files and an assembler to create class files. The Krakatau decompiler takes a different approach to most Java decompilers. It can be thought of more as a compiler whose input language is Java bytecode and whose target language happens to be Java source code. Krakatau takes in arbitrary bytecode and attempts to transform it to equivalent Java code. This makes it robust to minor obfuscation, though it has the drawback of not reconstructing the "original" source, leading to less readable output than a pattern matching decompiler would produce for unobfuscated Java classes.

Procyon [10] Updated in 2016. It handles language enhancements from Java 5 and beyond, up to Java 8, including enum declarations, enum and string switch statements, local classes (both anonymous and named), annotations, Java 8 lambdas and method references (i.e., the :: operator).

A. Tests

Different samples were tested in original researches, each of them provides specific area to test. The original research pointed out that all of the decompilers have their weak and strong sides, where none of them were able to decompile all of the samples [4] [5].

In our test we used samples from previous research [4] [11], but we recompile samples with current javac (version 1.8.0_131) or with current Jasmin (version 2.4) [6].

We devide test into two test groups:

Test group 1: Consist of samples used in previous research.

Test group 2: Consist of newly prepared samples with intent to test ability of actual decompilers to decompile new JDK features.

The following tests were in Test group 1:

Fibonacci [11] is a simple test sample for a decompiler. It writes out a Fibonacci number for a given number.

Casting [5] is a simple test sample to test decompiler ability to properly decompile char to int cast.

InnerClass [5] is a simple test sample to test decompiler ability to properly decompile inner classes.

TypeInference [23] is a test sample to test decompiler ability to deal with type inference for local variables.

Optimized [11] is a test sample generated by the Soot optimizer [24] on the TypeInference test program to test decompiler ability to properly decompile optimized byte code.

ControlFlow [23] is test sample to test decompiler ability to properly deal with handling of control flow.

Exceptions [23] is simple sample which contains two intersecting try-catch blocks. Although it is valid java bytecode, it wouldn't be produced by javac. This sample is created using Jasmin [6]. As mentioned in [11], the program used in the original tests [5] is incorrect, so a re-written version is used based on the call graph in original paper [23].

The following tests were in Test group 2:

StringSwitch is a sample taken from the official Oracle documentation. In the JDK 7 release, you can use a String object in the expression of a switch statement. The switch statement compares the String object in its expression with the expressions associated with each case label as if it were using the String.equals method; consequently, the comparison of String objects in switch statements is case sensitive. The Java compiler generates generally more efficient bytecode from switch statements that use String objects than from chained if-then-else statements. [34]

TryWithResources is another sample taken from official Oracle documentation. The try-with-resources statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements *java.lang.AutoCloseable*, which includes all objects which implement *java.io.Closeable*, can be used as a resource. [35]

Underscores in numeric literals are supported in Java from JDK7. Any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables us, for example, to separate groups of digits in numeric literals, which can improve the readability of the code. For instance, if the code contains numbers with many digits, we can use the underscore character to separate digits in groups of three in an analogous way how one would use a punctuation mark like a comma, or a space, as a separator. The sample was taken from the official Oracle documentation. [36]

Lambda expressions have been introduced in Java 8 and are touted to be the biggest feature of Java 8. Lambda expression facilitates functional programming and simplifies the development a lot. Sample is aimed to test decompilers' ability with handling lambda expressions. [37]

B. Results

Decompiler test results are shown in table II. Every decompiler was tested with all test samples mentioned in III.A. Sample were generated using javac, Soot, jasmin. The results have been evaluated using the effectiveness measure scale mentioned in II. Similarly, as in the previous research, none of the decompilers was able to decompile all samples correctly. Generally, we can sum up that newer decompilers perform much better in the tests.

Mocha [12], **SourceTec** [13], **jReversePro** [16] did not decompile any of the samples. They usually complain about unsupported class file versions.

SourceAgain [14] evaluation version is the only available through web. There are no other versions accessible. This makes this decompiler unqualified for our future research, so it was not processed in the results.

Jad [11] even though is really outdated, produces still quality results. It is probably the result of the chosen test samples, which did not covered newer Java features.

JODE [15] has produced satisfactory results. The bad thing is that even though it decompiled almost perfectly the majority of the samples, at the same time there were samples where it absolutely failed. Good point is that, for example, in testing the sample Exceptions it detected that try and catch blocks are intercepting. Also, it was the only decompiler from the original survey that correctly decompiled the sample Casting (without missing cast).

Dava [17] [18] results placed just behind Jad and JODE, where it produced similar balanced results to Jad. Although these results were not as good as from Jad, it performs really well on arbitrary bytecode produced by other tools but javac.

Jdec [19] with exception of Mocha, SourceTec, jReversePro which do not support current Java bytecode, performed probably the worst. It usually does not decompile bytecode produced by javac or arbitrary bytecode.

Java Decompiler [20] was the decompiler which outperformed the others in the original survey [11]. It has some troubles with decompiling arbitrary bytecode and also it missed together with the majority of the decompilers (Jad, Dava, Jdec, cfr, fernflower) cast statement in the sample Casting.

Cfr [7] is one of the new decompilers, which were not used in the original paper. Even though it outperforms most of the decompilers from the original survey, it finished worst from the new decompilers. Still it produces nice, good readable outputs.

Fernflower [8] produces quality outputs. Together with Cfr it has a problem with proper decompilation of the Casting sample and with arbitrary bytecode.

Krakatau [9] [22] is the only decompiler that was able to properly decompile the Exceptions sample. Its output is not always good readable, but in our tests it almost always produced syntactically and semantically correct output. This is probably the reason of a different approach to decompilation. Krakatau also states on its page, that its assembler and disassembler fully support Java 9, while the decompiler only supports Java 7. In particular, decompilation of lambdas is not supported. This was visible in results, that it wrongly decompiles 2 from 4 samples in test group 2.

Procyon [10] – although a newer decompiler which was not available during the original surveys, currently is not being developed. It produced the second-best results, with problems only with arbitrary bytecode produced by jasmin. As it has been mentioned, the good results of this little outdated decompiler are probably the result of the chosen test samples. Procyon developer also states that:

The Procyon decompiler handles language enhancements from Java 5 and beyond that most other decompilers don't. It also excels in areas where others fall short. Procyon in particular performs well with:

- Enum declarations;
- Enum and String switch statements (only tested against javac 1.7 so far);
- Local classes (both anonymous and named);
- Annotations;
- Java 8 Lambdas and method references (i.e., the: operator).

Results from the test group 2 have proved this, when it accomplished the best results.

Table 2 Decompiler tests results 1

	Fibonacci	**Casting**	**InnerClass**	**TypeInference**	**Optimized**	**ControlFlow**	**Exceptions**
Mocha	9	9	9	9	9	9	9
SourceTec	9	9	9	9	9	9	9
SourceAgain	0	5	8	1	3	1	7
Jad	0	5	2	1	4	1	4
JODE	0	0	9	0	2	1	9
jReversePro	9	9	9	9	9	9	9
Dava	0	5	8	2	2	1	9
Jdec	9	5	8	8	8	7	8
Java Decompiler	0	5	0	2	3	7	8
Cfr	0	5	0	2	2	0	6
Fernflower	0	5	0	0	0	0	9
Krakatau	0	0	2	0	0	0	0
Procyon	0	0	0	0	0	0	6

Fig. 1 together with Fig. 2 depict graphically the research outputs based on test group 1. Fig. 1 shows graphically how effectively are the decompilers able to deal with arbitrary and javac generated bytecode. There are their total scores based on defined metrics (lower result is better). It is clear that newer decompilers perform better and also decompilers in general have bigger problems with decompiling arbitrary byte code rather than bytecode generated by javac.

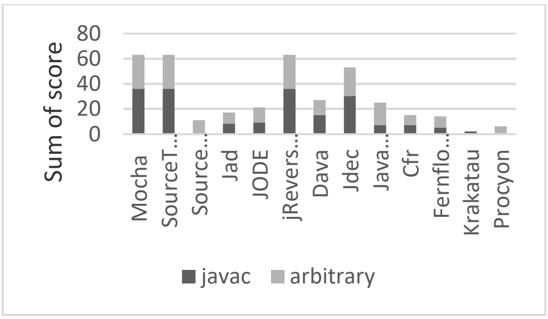

Fig. 1 Decompiler effectiveness on test group 1

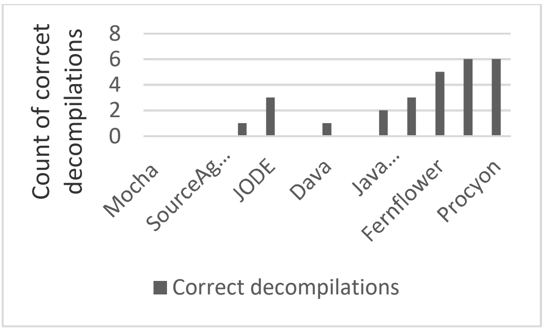

Fig. 2 Correct decompilations of test group 1

Fig. 3 together with Table 3 shows how the decompilers preform with the test group 2. As in the previous results, the performance of all decompilers is the same. As it was visible in the previous table, Procyon showed the best results.

Table 3 Decompiler tests results group 2

	String Switch	**Try With Resources**	**Underscores**	**Lambda**
Cfr	1	1	1	0
Fernflower	1	1	1	1
Krakatau	1	9	1	5
Procyon	0	0	2	0

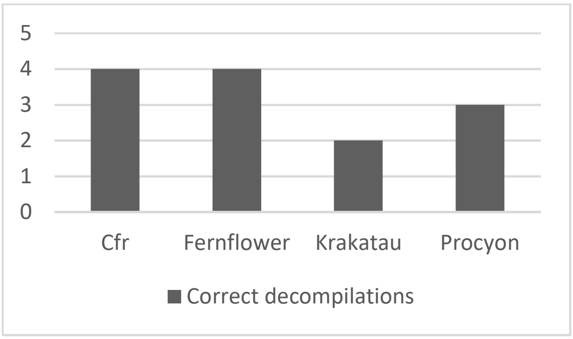

Fig. 3 Correct decompilations of test group 2

C. Comparison of results

As it was depicted in table III., since the original survey conducted in 2003 [5] and 2009 [4], none of the decompilers, with the exception of Java Decompiler, did not get an update. There are four new decompilers. These new decompilers (Cfr, Fernflower, Krakatau and Procyon) almost completely outperformed all the decompilers from the mentioned surveys. This is absolutely visible in fig. 3 as the average score of the decompilers used in the original survey, without those which do not support the current java bytecode (Mocha, SourceTec, jReversePro) and also without SourceAgain (since only web version is available, which is not suitable for our further research) is 4,08, while the average score of the current decompilers is 1,3 and overall average score is 4,33.

Table 4 Decompilers

Decompiler	2003 version [5]	2009 version [4]	Current version	Last update
Mocha	0.1b	0.1b	0.1b	1996
SourceTec	1.1	1.1	1.1	1997
SourceAgain	1.10j	1.1	1.1	2004
Jad	1.5.8e	1.5.8e	1.5.8e	2001
JODE	unknown	1.1.2-pre1	1.1.2	2004
jReversePro	1.4.1	1.4.2	1.4.2	2005
Dava	2.0.1	2.3.0	Nightly build from 20-05-2017	Daily nightly builds
Jdec	N/A	2.0	2.0	2008
Java Decompiler	N/A	0.2.7	0.7.1	2014
Cfr	N/A	N/A	0.122	2017
Fernflower	N/A	N/A	Master branch from 10.5.2017	2017
Krakatau	N/A	N/A	Master branch from 10.5.2017	2017
Procyon	N/A	N/A	0.5.30	2015

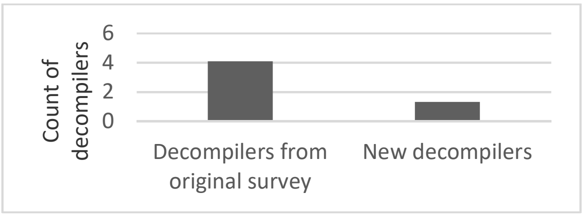

Fig. 4 Comparison between decompilers from original survey and current decompilers

Even though several years have passed since the original survey [4] was published, overall evaluation is roughly the same. This might be caused by the chosen test set, since the same samples have been used. Especially on the side of the decompilers which were previously classified as the weakest, there were no differences. They could not decompile any of the test samples. They usually do not support current *class* files.

On the other side of the spectrum, still no decompiler was able to decompile all the test samples correctly. The best results have been achieved by Procyon and Krakatau. They were able to decompile 6 of 7 samples correctly. Newer decompilers have significantly higher success rate and quality of the decompiled outputs.

When we focus our attention on the test group 2, we can see that almost all up-to-date decompilers, with the exception of Krakatau, are able to handle successfully all new samples. They generally produce output of high quality when used on *javac* compiled bytecode.

4 CONCLUSION AND FUTURE WORK

This paper was prepared with the intent to evaluate current state of art of Java decompilers. We are planning to make Hybrid analysis Android application framework, where we want to combine information gathered using static and dynamic analysis to detect Android malware and analyze its behavior.

Decompilation is one of the main parts of static analysis. To properly determine all execution paths of a sample, it is crucial to properly decompile it. As it was visible in table II., even if the decompiler produces syntactically correct output, it does not always necessary mean that it is also semantically and logically correct. If the analyst during the analysis relies on these wrong outputs, he might, for example, miss some of the malicious functionality. For these reasons, it is essential to properly know the tools and not just blindly trust to their outputs. Quick Android Review Kit (QARK) [25] solved the mentioned problems by an approach in which it applies multiple decompilers on the sample and then merges their outputs.

The results are similar to the results of the previous surveys [4] [5]. Due to this fact, new test samples containing modern Java features like string enum switches or lambda functions and others were prepared. This is the main extension from the previous research published in [38].

During the Android malware analysis itself, another step came into consideration. It is the Dalvik bytecode into Java bytecode transformation. There are tools like enjarify [26] and dex2jar [27] that can handle those transformations. Also, the evaluation of the outputs of these tools would seem to be valuable. Respectively comparison of outputs from combinations of these converting tools together used with Java bytecode decompiler with outputs from direct DALVIK bytecode decompiler such as jadx (free, open source) [28] or jeb [29] (commercial) would be also valuable.

Last, but not least, this year, during the Google I/O conference [30], support for Kotlin language for Android application development [31] [32] has been announced. Therefore, it would be worthwhile to test the ability of current tools to analyze malware developed in Kotlin.

References

[1] EILAM, E., CHIKOFSKY, E. J. *Reversing: secrets of reverse engineering.* Indianapolis : Wiley, c2005.

[2] SIKORSKI, M., HONIG, A. *Practical malware analysis: the hands-on guide to dissecting malicious software.* San Francisco : No Starch Press, c2012.

[3] SCHMIDT, A. D., SCHMIDT, H. G., BATYUK, L., CLAUSEN, J. H., CAMTEPE, S. A., ALBAYRAK, S., YILDIZLI, C. *"Smartphone malware evolution revisited: Android next target?"* in 2009 4th International Conference on Malicious and Unwanted Software (MALWARE). Montreal : 2009.

[4] HAMILTON, S. D. James. *An evaluation of Current Java Bytecode Decompilers.* IEEE International Workshop on Source Code Analysis and Manipulation. IEEE Computer Society, 2009. pp. 129-136.

[5] EMMERIK, M. V. "Java decompiler tests", 2003. [Online]. Available: <http://www.program-transformation.org/Transform/JavaDecompilerTests>.

[6] MEYER, D. R. Jonathan: "*Jasmin*", 2004. [Online]. Available: <http://jasmin.sourceforge.net/>. [Accessed 01 04 2017].

[7] BENFIELD, L. "*CFR*", 12 06 2017. [Online]. Available: <http://www.benf.org/other/cfr/index.html.> [Accessed 12 06 2017].

[8] USHAKOV, E. "fernflower", 8 6 2017. [Online]. Available: <https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine>. [Accessed 8. 6. 2017].

[9] GROSSE, R. "*Java decompiler, assembler, and disassembler*", 17 05 2017. [Online]. Available: <https://github.com/Storyyeller/Krakatau>. [Accessed 22. 05. 2017].

[10] STROBEL, M. "*Procyon*", 18 08 2016. [Online]. Available: <https://bitbucket.org/mstrobel/procyon/>. [Accessed 03 06 2017].

[11] HAMILTON, J. "Decompiling Java", 06 05 2009. [Online]. Available: <https://jameshamilton.eu/sites/default/files/DecompilingJavaWorkingDocument.pdf>. [Accessed 06 05 2017].

[12] VLIET, H. Mocha, the java decompiler, 1996.

[13] S. S. Inc., "Sothink Java Decompiler," [Online]. Available: <http://www.sothink.com/product/>.

[14] SOFTWARE, A. "SourceAgain". [Online]. Available: <http://www.ahpah.com/cgi-bin/suid /~pah/demo_>.

[15] HOENICKE, J. "JODE", 06 05 2013. [Online]. Available: <https://sourceforge.net/projects/ jode/>. [Accessed 03 05 2017].

[16] KUMAR, K.:"JReversePro - Java Decompiler," 08 04 2013. [Online]. Available: https://sourceforge.net/projects/jrevpro/. [Accessed 22 03 2017].

[17] "Dava: A tool-independent decompiler for Java", [Online]. Available: <http://www.sable.mcgill.ca/dava/>.

[18] MIECZNIKOWSKI, J. New algorithms for a Java decompiler and their implementation in Soot. Master thesis. Quebec : 2003.

[19] SWAROOP, B., BETTADAPURA, K. "Jdec: Java decompiler", 18 07 2013. [Online]. Available: <http://jdec.sourceforge.net>. [Accessed 03 03 2017].

[20] DUPUY, E. "Java Decompiler", 25 03 2015. [Online]. Available: <http://jd.benow.ca/.> [Accessed 10 03 2017].

[21] CHEPTSOV, A. "IntelliJ IDEA 14 EAP 138.1029 is out with a built-in Java decompiler", 11 07 2014. [Online]. Available: <https://blog.jetbrains.com/idea/2014/07/intellij-idea-14-eap-138-1029-is-out/>. [Accessed 11 05 2017].

[22] PROEBSTING, T. A., WATTERSON, S. A. Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?), Proc. Third USENIX Conf. Object-Oriented Technologies and Systems (COOTS), 1997.

[23] MIECZNIKOWSKI, J., HENDREN, J. L. "Decompiling java bytecode: Problems, traps and pitfails," CC '02: Proceedings of the 11th International Conference on Compiler, pp. 111-127, 2002.

[24] BODDEN, E. "Soot", [Online]. Available: <https://sable.github.io/soot/>.

[25] LinkedIn, "qark". [Online]. Available: <https://github.com/linkedin/qark>.

[26] "enjarify" [Online]. Available: <https://github.com/Storyyeller/enjarify>.

[27] PAN, B. "Dex2jar: Tools to work with android. dex and java. class files", [Online]. Available: <https://sourceforge.net/projects/dex2jar/>.

[28] "jadx - Dex to Java decompiler," [Online]. Available: <https://github.com/skylot/jadx>.

[29] "JEB: Android Decompiler + Android Debuggers", [Online]. Available: <https://www.pnfsoftware.com/jeb2/>.

[30] "Google I/O is an annual developer festival held at the outdoor Shoreline Amphitheatre. See you next year!" [Online]. Available: <https://events.google.com/io/>.

[31] SHAFIROV, M. "Kotlin on Android. Now official", 17 05 2017. [Online] Available: <https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/>.

[32] "Kotlin and Android". [Online] Available: <https://developer.android.com/kotlin/index.html#kotlin-android-support-announced-at-google-io>.

[33] DUPUY, E. "JD Project", [Online]. Available: <http://jd.benow.ca/>. [Accessed 01 06 2017].

[34] "Strings in switch Statements" [Online]. Available: <https://docs.oracle.com/javase/8/docs/technotes/guides/language/strings-switch.html>.

[35] "The try-with-resources Statement" [Online]. Available: <https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>.

[36] "Underscores in Numeric Literals" [Online]. Available: <https://docs.oracle.com/javase/8/docs/technotes/guides/language/underscores-literals.html>.

[37] "Java 8 - Lambda Expressions" [Online]. Available: <https://www.tutorialspoint.com/java8/java8_lambda_expressions.htm>.

[38] KOSTELANSKÝ, J., DEDERA, L. "An evaluation of output from current Java bytecode decompilers: Is it Android which is responsible for such quality boost?" *2017 Communication and Information Technologies (KIT).* Vysoke Tatry : 2017. pp. 1-6.doi: 10.23919/KIT.2017.8109451

Eng. Jozef KOSTELANSKÝ
PhD. Student
Armed Forces Academy of General M. R. Štefánik
Department of Informatics
Demänová 393
031 06 Liptovský Mikuláš
Slovak Republic
E-mail: jozef.kostelanskyw@gmail.com

Assoc. Prof. RNDr. Ľubomír DEDERA, PhD.
Armed Forces Academy of General M. R. Štefánik
Department of Informatics
Demänová 393
031 06 Liptovský Mikuláš
Slovak Republic
E-mail: lubomir.dedera@aos.sk

Eng. Jozef Kostelanský graduated from Armed Forces Academy, Liptovsky Mikulas in Computer system, networks and services in 2016. Currently he continue with doctoral studies also at the Department of Informatics, Armed Forces Academy in Liptovský Mikuláš, Slovakia. His research interests include the computer security.

Assoc. Prof. RNDr. Ľubomír Dedera, PhD. graduated from the Comenius University, Bratislava in Computer Science in 1990. Currently he works as an associate professor at the Department of Informatics, Armed Forces Academy in Liptovský Mikuláš, Slovakia. His research interests include the area of computer languages and computer security.

NTSP 2018
NEW TRENDS IN SIGNAL PROCESSING
14TH INTERNATIONAL CONFERENCE
ON NEW TRENDS
IN SIGNAL PROCESSING 2018

NEW TRENDS IN SIGNAL PROCESSING 2018

October 10 – 12, 2018

in Hotel Chopok, Demänovská dolina, Slovakia

The conference covers main topics:

- **Signal Processing**
- **Applied Electronics**
- **Information and Communication Engineering**
- **Microwave Engineering**
- **Signal Processing in Military Applications**

All papers for the NTSP 2018 will be reviewed and published in electronic form on DVD with **ISBN 978-80-8040-546-5** and **ISSN 1339-1445** and will be submitted to **IEEE Xplore database**.

Organizers: Armed Forces Academy of General M. R. Štefánik, Liptovský Mikuláš, Department of Electronics and The Slovak Elektrotechnic Society – affiliated branch Liptovský Mikuláš.

For more information see the conference website: **http://ntsp2018.aos.sk/**