# Protecting Java Applications through Bytecode Transformations

Thomas Spranghers, Wim Vander Schelden

Promotors: Bjorn De Sutter, Koen De Bosschere

*Abstract*— **Since bytecode-based languages have become dominant in software engineering, they have also become an interesting target for analysis, reverse engineering and other attacks. In contrast to machine language, bytecode often contains a large amount of metadata which can be used to recover the original design of an application and many aspects of the software engineering process. This means that many attacks become significantly easier when compared to their machine language equivalents. Over the past year, we have attempted to make bytecode applications more secure by researching transformations on bytecode and its metadata to hide, remove or obfuscate the familiar building blocks of software engineering.**

*Keywords*— **protection, Java, obfuscation, bytecode**

## I. Introduction

OVER the past 20 years, bytecode-based programming languages have become more and more popular in software engineering [1]. The best-known example is the Java platform, and more recently other major players have developed bytecode-based environments as well, such as Microsoft's .NET platform or Adobe's Flash.

While these languages often offer interesting features for developers, such as automatic memory management, an extensive standard library and reflection, some of these features come at a cost in terms of security. To support more advanced programming techniques, it is often necessary to embed extra information about the source code in the program's bytecode. This information can also be used to uncover the structure of the application by potential attackers. For instance, it is trivial to reconstruct the classes and their relations. Such knowledge can then be used to study the inner workings of these classes, or to get a better understanding of the design of the application as a whole. This is often a first step in finding security holes in a program and other attacks.

In order to prevent such attacks, we have chosen to develop a set of bytecode transformations to remove, hide or otherwise obfuscate the structure of an application. We work on a bytecode level, as opposed to on the raw source code. This is because it simplifies the problem domain a great deal, and because it allows programmers to develop and debug applications without any extra hassle. We've also chosen to focus on the Java platform specifically, mainly because it has more existing research in and tools for this topic for us to build on.

## II. Technology overview

There is already a sizeable collection of existing work for securing bytecode applications. Many transformations have already been researched and developed, and some of them are available in commercial products.

The most common transformation is the renaming of classes, methods and fields [2]. While this doesn't change anything about the structure of the application, it does effectively remove almost all human-readable information from the class files. A frequently used extension of this technique is name overloading [2, 3]: using the same name over and over, as long as it doesn't cause ambiguous situations.

One of the more complex transformations involves modifying the control flow within the application [4]. This is achieved by adding conditional constructs and loops using opaque predicates [5] to an application, which don't alter the behaviour of the application, but significantly complicate the control flow graph.

Since novice attackers often resort to decompilation, it is also an option to make sure the bytecode no longer has a simple translation to source code. Such a translation can be prevented by using characters and words which have a special meaning at source code level, but don't have any role at bytecode level [2, 3]. Using such keywords or characters as identifiers assures developers that decompilation will be hindered significantly.

## III. Architecture

The global architecture of the obfuscation framework is based on a modified "pipes and filters" pattern [6], as shown in Figure 1. A central coordinating component is added to offer functionality not available in the traditional pattern, such as operations not easily performed on a stream of data. By making each transformation a filter, this pattern allows us to easily make transformations modular and chain them together to create increasingly complex bytecode programs.

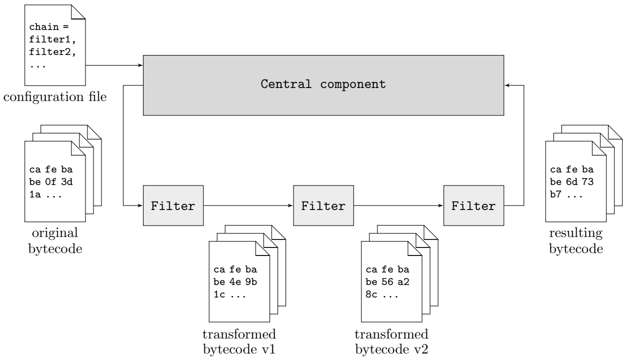

Fig. 1: Architecture of the obfuscation framework.

Each filter consists of the following four phases:

- The **analysis phase**, where the bytecode is sent through the filter a first time. Useful information is stored for later processing.
- The **processing phase** uses the information collected in the previous step to calculate any derived properties where needed.
- The **transforming phase** applies the actual transformations to the bytecode, using the information collected in the previous steps.
- The **finishing phase** is used to add additional library classes or remove classes which are no longer needed by the program after the transformations.

## IV. Implementation

When deciding on what transformations would be most interesting to research and evaluate, we focused on transformations with the most potential to alter the overall structure of an application. This excluded techniques which focused on altering the inner control flow of a single method.

Inlining methods was the first transformation we developed. It removes private methods, copying their code to wherever they are used. This makes it harder to analyze and match different uses of the methods.

A second transformation removes the package structure from the application whenever possible, by placing all classes into a single artificial package. Analysing such a package becomes more complicated, because the meaningful segmentation of the application has been removed.

A similar technique was applied to classes: merging classes with a completely different function and behaviour creates a class where the behaviour is not clearly visible.

The most challenging transformation we researched was the use of runtime modifications in Java. While our implementation is restricted in several ways by the JVM, we succeeded in transforming an application to modify itself during startup. The specifics of this filter also generate a large number of additional classes, and it complicates the inheritance tree significantly.

## V. Evaluation

The evaluation of the obfuscation is two-fold: first of all, you want a more secure application, and second, you want this extra security without slowing down your application significantly. To measure the slowdown, we obfuscated SPECjvm2008 [7] and measured the speed of its benchmarks. Figure 2 shows the results, compared to the speed without obfuscation. The performance is measured in operations per minute. As you can see from the image, no significant performance hits were observed.

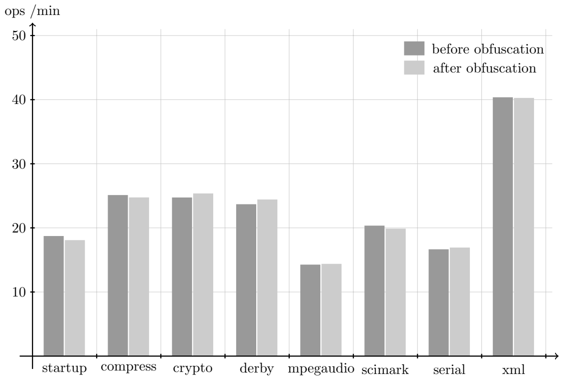

Fig. 2: SPECjvm2008 results.

Measuring application security is less straightforward. To compare the complexity of an application before and after obfuscation, we use established software package metrics [8]. The most important of these metrics is the distance from the Main Sequence. It describes how far away the package is from the ideal design for software, where 0 would indicate a perfectly designed application and 1 would indicate the total lack of structure.

As can be read from figure 3, our transformations consistently move packages further away from this perfect design ideom. The horizontal axis displays the number of classes in the test projects, a set of simple real-world applications, grouped into categories. Unfortunately, we were unable to compare these results with other obfuscators as they were unable to process any of our test programs correctly.

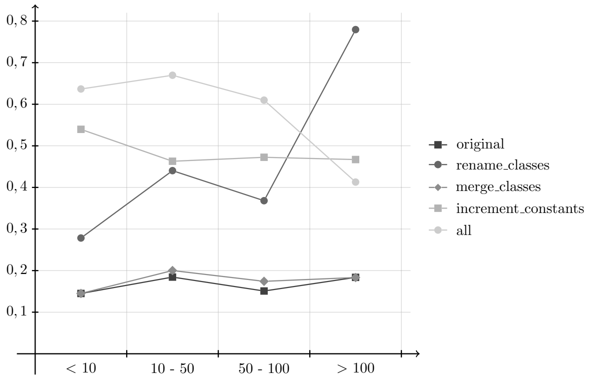

Fig. 3: Distance from the Main Sequence.

## VI. Conclusion

We set out to research and develop bytecode transformations that would make an application as a whole harder to understand. Using the transformations described above, we managed to significantly increase the complexity of the structure of a set of real-world applications.

## References

[1] “TIOBE programming community index,” `http://www.tiobe.com/index.php/content/paperinfo/tpci/`.

[2] S. Cimato, A. De Santis, and U. Ferraro Petrillo, “Overcoming the obfuscation of Java programs by identifier renaming,” *J. Syst. Softw.*, vol. 78, no. 1, pp. 60–72, 2005.

[3] Jien-Tsai Chan and Wuu Yang, “Advanced obfuscation techniques for Java bytecode,” *J. Syst. Softw.*, vol. 71, no. 1-2, pp. 1–10, 2004.

[4] Douglas Low, “Java control flow obfuscation,” Tech. Rep., 1998.

[5] Ginger Myles and Christian Collberg, “Software watermarking via opaque predicates: Implementation, analysis, and attacks,” *Electronic Commerce Research*, vol. 6, no. 2, pp. 155–171, 2006.

[6] Regine Meunier, “The pipes and filters architecture,” pp. 427–440, 1995.

[7] “SPECjvm2008 benchmarks,” `http://www.spec.org/jvm2008/`.

[8] Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, 2002.