OBLIVE: Seamless Code Obfuscation for Java Programs and Android Apps

Davide Pizzolotto*†, Roberto Fellin*†, Mariano Ceccato*
*Fondazione Bruno Kessler, Trento, Italy
†University of Trento, Trento, Italy

***Abstract*—Malicious reverse engineering is a problem when a program is delivered to the end users. In fact, an end user might try to understand the internals of the program, in order to elaborate an attack, tamper with the software and alter its behaviour. Code obfuscation represents a mitigation to these kind of malicious reverse engineering and tampering attacks, making programs harder to analyze (by a tool) and understand (by a human).**

In this paper, we present *Oblive*, a tool meant to support developers in applying code obfuscation to their programs. A developer is required to specify security requirements as single-line code annotations only. *Oblive*, then, reads annotations and applies state-of-the-art data and code obfuscation, namely xor-mask with opaque mask and java-to-native code, while the program is being compiled. *Oblive* is successfully applied both to plain Java programs and Android apps.

Showcase videos are available for the code obfuscation part https://youtu.be/Bml-BkKP3CU and for the data obfuscation part https://youtu.be/zUizYVK42ps.

***Index Terms*—Program transformation; Code obfuscation; Data obfuscation**

I. Introduction

Java applications and Android apps are shipped in bytecode format in order to be executed by the Java/Dalvik Virtual Machine. However, bytecode is quite easy to read and decompile, much easier than plain binary machine code, and this exposes a program to Man-at-the-end (MATE [3]) attacks. These attacks are those mounted by an end user, with the aim of applying malicious reverse engineering or tampering with the code or with the program data.

Code obfuscation is becoming more and more widely adopted [11] by transforming the original program such that attacker analysis becomes harder. This harder analysis mitigates the possibility to be attacked, by turning it impractical or economically infeasible.

In this paper, we present *Oblive*, a tool aimed at helping developers in effectively and quickly applying code obfuscation, without wasting their time in configuring or applying complex obfuscation tools. The novel features of *Oblive* are:

- *Seamless integration with the build process: Oblive* is developed on top of Gradle, a mainstream build system, used in most open source programs (especially Android apps). As a Gradle plugin, it can intercept the Gradle build pipeline and extend it exactly where needed. It injects two additional build tasks, that perform respectively data obfuscation and code obfuscation.
- *Obfuscation configuration with code annotations:* The developer is left with the only manual task of deciding which part of code is the critical one, that should be obfuscated. This decision is specified by adding Java annotations, directly in the clear source code.
- *Cutting edge obfuscations: Oblive* contains two components, based on existing obfuscation approaches available in literature. Data obfuscation includes recent variants of *Xormask* [8], that turns static analysis very hard. Code obfuscation replaces Java methods with an equivalent implementation in C [7], much harder to analyze.

This paper is structured as follows. Section II describes the motivations behind the need of *Oblive*. We describe the implementation and architecture of *Oblive* in Section III. The assessment of resilience and correctness of obfuscated code is covered in Section IV. Then, Section V compares *Oblive* with the state of the art, and Section VI closes the paper.

II. Motivation

Many business sectors (especially apps) require software artifacts to be delivered with a fast time-to-market. In these contexts, the development and deployment process should be simple and smooth. Complex or external tools might represent a threat to a fast delivery of new products or updates. While obfuscation would be highly beneficial, it would be adopted only when it would guarantee fast deployment. Thus a first requirement for an obfuscation tool is:

> ***Requirement Req*$_1$***: Obfuscation tools should be seamlessly integrated in the program build process and should not be run manually by developers.*

Oblive meets this requirement, because it is implemented as a plugin for Gradle, one of the mainstream build environments for Java and Android. To integrate our obfuscation in an existing Java or Android project, a developer just needs to specify a dependency (one line to edit) in the main Gradle build script. Our plugin would be automatically downloaded at the first build (as Gradle plugins normally are) and cached locally for future builds.

Some tools indeed do not require manual effort to apply, for instance ProGuard[1] comes with AndroidStudio[2], the

[1]https://www.guardsquare.com/en/products/proguard
[2]https://developer.android.com/studio/

mainstream IDE for Android Apps. However, even if fast to apply, this tool only supports quite basic obfuscations, such as *identifier renaming*, that are quire established in literature since some years [6]. The most recent obfuscation algorithms [8], [7] are only available as separate tools. So an additional requirement is the following:

> ***Requirement Req***$_2$*: Obfuscation tools should implement advanced state-of-the-art obfuscation algorithms.*

Oblive satisfies this requirement by implementing two of the most recent and resilient data and code obfuscation algorithms, whose effectiveness against malicious reverse engineering has been assessed [2].

A last feature that implies fast adoption of an obfuscation tool, is to be simple to use. In fact, often complex configuration files need to be written in a domain specific language (such as ProGuard configuration files) that developers might not be familiar with. Alternatively, even more complex command-line options might be needed to drive and configure obfuscation algorithms.Thus, the last requirement is:

> ***Requirement Req***$_3$*: Obfuscation transformations should be configured with a language that is familiar to developers, i.e. not a domain-specific language and not command-line options.*

Oblive meets this requirement by relying on Java annotations to specify obfuscation parameters. These annotations are pure Java code, which is familiar to Java developers. In fact, Java annotations are largely used to configure mainstream tools and libraries, such as the Spring framework[3] with its many submodules. Moreover, common IDEs often provide support when writing annotations with code completion, suggestions and syntax checking.

In the rest of the paper, we will show how *Oblive* practically meets all these requirements, to deliver a novel, effective and easy way to integrate a catalog of obfuscation transformations.

III. Transformation

In this section, we describe the architecture of *Oblive*. With the aim of helping software developers in adopting state of the art obfuscation simply and quickly, our tool is based on source code annotation. *Oblive* automatically applies obfuscation transformations to the compiled bytecode. It has been developed as a Gradle plugin, in order to apply both to Android apps and to Java desktop developers.

Oblive contains two main components that integrate two obfuscation algorithms. They are data obfuscation with *Xormask* [8], and code obfuscation with *Java2c* [7]. These two obfuscation algorithms have have been extensively described in previous works, however, we summarize them here for completeness.

A. Data Obfuscation: Xormask

The *Xormask* component is used to hide sensitive values of class fields. Values are encoded with a mask, so that the clear value of the field is difficult to guess using malicious reverse engineering attacks based on static analysis. The value is encoded using a bitwise xor mask. The value is decoded as needed, thus limiting the windows of exposure of the sensitive clear value.

The *Xormask* component can be applied in three different variants, each one with increasing level of obscurity, at the cost of increased execution overhead. The simplest variant is the *Constant* xormask, where the mask is a static constant stored in the code. This variant provides the least resilience but its runtime impact is almost negligible. The most advanced variant, the *Opaque* xormask, requires the attacker to solve an NP-hard problem using K-clique [7] every time the variable is accessed in order to get the mask value. A trade-off between the two is represented by the *Opaque with cache* xormask, which uses the same K-clique NP-hard problem, but the solution is computed only once and then stored in a cache for fast access. It is less secure, because an attacker could also access the cache to retrieve the mask and decode the clear value.

B. Code Obfuscation: Java2c

The *Java2c* component is used to hide an entire method body, by translating it to native code (i.e., C code that is later compiled into a native library) that will be dispatched together with the application. As such it is subject to the Java constraints that prevents constructors to be native, but any other Java method can be obfuscated with this approach.

The *Java2c* component reads the Java code, opcode by opcode, and emit their C counterpart implementation, essentially emulating the JVM in the generated C code. The C implementation is written in such a way that it is still possible to query the JVM for regular Java method calls, exception handling and field access, thus granting the ability of translating every Java opcode and preserving the semantics of the overall program.

C. Plugin Integration

To integrate *Oblive* as a Gradle plugin, the developer has to specify a dependency in the main Gradle build script (the *build.gradle* file) as shown in Figure 2. This will make Gradle load our plugin, that specifies how to change the build process to apply code and data obfuscation at the right build phase, i.e. just after compiling the source code into Java bytecode.

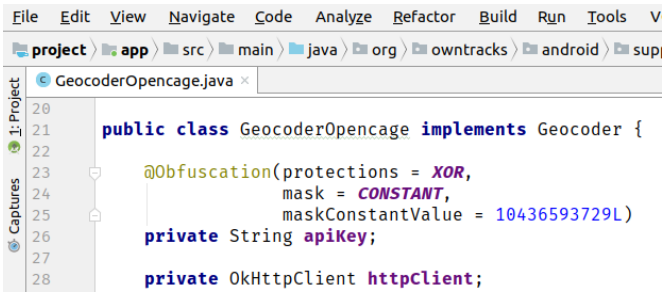
Then, obfuscated code will continue with the remaining build process, possibly edited by other plugins. For instance, the *Android* plugin requires the Java bytecode to be converted to Dalvik bytecode, be packaged into an APK archive and be cryptographically signed.

D. Annotation Syntax

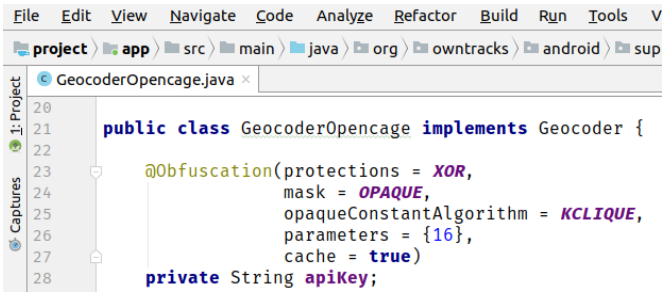
Oblive requires code annotations in order to apply code or data obfuscation. The annotation is `@Obfuscate` with some parameters that depend on the particular code/data obfuscation to be deployed.

Here we present the parameters for the data obfuscation. Figure 1 shows an example where the string variable `apiKey`

[3]https://spring.io/



(a) Xormask with constant mask obfuscation



(b) Xormask with opaque mask obfuscation

Fig. 1: *Examples of code annotation for data obfuscation with Xormask.*

```
...
apply plugin: 'eu.fbk.plugin.hardening'
...
```

Fig. 2: *The Gradle snippet that loads* Oblive

is obfuscated with a constant mask on the left and with an opaque mask on the right. The `protections` parameter specifies which kind of obfuscation we want to apply, in this case the xormask. `mask` specifies the xormask variant to use. If the variant chosen is `CONSTANT`, `maskConstantValue` specifies the mask static constant value. If the variant chosen is instead the `OPAQUE` one, `opaqueConstantAlgorithm` specifies the algorithm used to generate the opaque constant and `parameters` sets a parameter that corresponds to the size of the hard problem used to generate the opaque constant. Then, `cache` is a boolean representing if the problem should be solved just once and then cached. Note that the larger the problem size, the higher the data obscurity, but also the slower the code. Thus we recommend to keep a value of 16. In order to simplify the usage of *Oblive*, most of these values are defaulted to recommended ones.

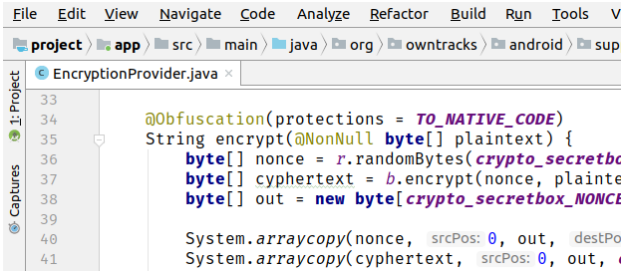


Fig. 3: *Example of code annotation for code obfuscation with Java2c.*

Figure 3 shows an example of the code obfuscation where the *Java2c* obfuscation is applied to the method `encrypt`. For code obfuscation with *Java2c*, the parameter `protections` is the only parameter to specify.

E. Transformation

Oblive is implemented in the form of a Gradle plugin defining two novel Gradle tasks, called *xormask* and *java2c*, named after the two components that obfuscate the bytecode. The Gradle plugin takes the Java bytecode generated by the `compileJavaCodeWithJavac` Gradle task and applies the two obfuscation tasks to it, thus modifying the generated bytecode files.

When the *Java2c* transformation is applied, an additional task that compiles the C code starts at the end of the plugin obfuscation. The generated C binary files are ready to continue with the Gradle pipeline in a transparent way. Then, the build continues with the ordinary Java or Android tasks, and the packaged app/application (apk or jar file) is automatically generated.

It is worth noting that the plugin ensures that the two obfuscation tasks are applied in the aforementioned order due to the inability of the *Xormask* component of obfuscating native code produced by the *Java2c* component.

IV. Empirical Validation

Both *Xormask* and *Java2c* components of *Oblive* have been tested and evaluated especially in terms of execution overhead of obfuscated code [8], [7]. In this paper, instead, we focus on assessing the correctness of the generated code and the automation of the obfuscation process.

A. Case Studies

For our empirical evaluation, we collected several open source Android apps, by searching those available as open source in GitHub[4]. Our initial list of apps has been filtered to keep only those for which the test cases are available, and that use Gradle as a build system.

In our experimental validation, we considered three open source Android apps:

- *cyclestreets/android*, a journey planner system;
- *owntracks/android*, a tracking application;
- *duckduckgo/android* a search engine.

B. Correctness

In order to assess the correctness of *Oblive*, we first ran the entire test suite on the clear version of the case study Android projects. We, then, proceeded to obfuscate with *Java2c* the 10 longest methods of each case study, with methods selected

[4]https://github.com/

among those covered by test cases. Additionally, for each class containing these methods, we also obfuscated with *Xormask* a random field among the most used.

Then, we executed the obfuscated code on two Android devices, one with an x86 processor and one with an ARM processor. We observed that every test passing in the clear code was also passing in the obfuscated code. Thus, we can conclude that *Oblive* preserves the semantics of the obfuscated code of these case studies.

C. Resilience to Manual Inspection

In order to test obscurity of the code, we decompiled the application with the `javap` bytecode viewer, part of the Orcale Java Development Kit, and with the *CFR* Java decompiler[5], to manually verify if the produced bytecode was indeed obfuscated.

In case of code obfuscation, both *javap* and *CFR* showed just the native signature of the method. Additionally, per *Java2c* implementation, also the constant strings are not written in the `.rodata` section of the generated native library, thus increasing resilience.

In matter of data obfuscation with the opaque mask, *javap* and *CFR* showed the NP-hard problem required to collect the data mask. In particular, the latter tried but failed at reconstructing the problem in plain Java code, generating around a thousand lines of code for every use of the obfuscated field. In all the cases, including the most simple with a static constant mask, the clear value of the field was hidden.

D. Resilience to Automated Analysis

Finally, we also studied the resilience of the code generated by *Oblive* against automatic analysis by tools, in particular with respect to malware detection. In particular, we used one of the most popular collection of malware detection kits, i.e. *VirusTotal* malware scanner[6]. Although hiding malware is not the intended use of our tool, this is an effective way to measure the impact of obfuscation with an analysis tool that applies with a very specific analysis goal.

We took a malicious app from the Drebin malware dataset [1], namely *BlackListPro* and we decompiled it. We manually inspected the decompiled sources to understand the malicious behavior. Then, we implemented a piece of proof-of-concept malware with a similar behaviour, to be used in our empirical evaluation. This consisted in reading data from the filesystem and then sending part of it as SMS to an hardcoded destination number.

We submitted this app to the *VirusTotal* online scanner. Out of 60 services available in *VirusTotal*, 21 of them reported malware or unwanted behaviour in our app.

Then, we applied code obfuscation to the method that sends the SMS, and we applied data obfuscation to the variable holding the phone number, destination of this SMS. After submitting the obfuscated app for scanning, only 1 out of 60 services detected our obfuscated app as malware.

[5]http://www.benf.org/other/cfr/

[6]https://www.virustotal.com

The scan results are reported in Figure 4, they are the two screenshots of the virus scan on respectively the clear application and the obfuscated application.

We can thus assess that our obfuscation increases the resilience against automated code analysis.

V. Related Work

Obfuscation is used to make code obscure, so that it is more complex to understand by a potential attacker, who wants to reverse engineer it. Obfuscation techniques change code structure without changing its functional behavior through different kinds of code transformations [10], [9]. It is well-known that for binaries that mix code and data, disassembly and de-compilation are undecidable in the worst case [5].

Data obfuscation targets data and data structures contained in the program. *Oblive* automates our previous approach [8], that aims at encoding program values using a mask that is quite difficult for static analysis to recover, because it would require to solve a computationally expensive hard problem.

The idea of translating Java bytecode to C was investigated by Hsieh et al. [4] with the aim of making a program execute faster, and by us [7] to turn malicious reverse engineering more difficult and to enable more efficient obfuscations. We reused the latter approach in *Oblive*, with a much higher level of automation.

Other obfuscation tools for Java and Android are available and the most prominent ones are *SandMark*[7], *Allatori*[8], *Zelix*[9], *DexGuard*[10]. Among them, *SandMark* is the only academic one. It includes proof-of-concept implementations of novel obfuscations elaborated by its academic authors. However, the tool is not very easy to configure, because each obfuscation accepts a different set of command line arguments, to be properly configured.

Allatori, *Zelix* and *DexGuard* are instead quite robust commercial tools, but they implement quite established obfuscations [11]. *Oblive*, instead, implements novel cutting-edge research obfuscations [8], [7].

Similarly to *SandMark*, also *Allatori* obfuscations can be configured using command line parameters. Instead of relying on (potentially quite complex) command line parameters, *Oblive* aims at supporting highest productivity by adopting code annotations. In fact, while command line options require to be manually checked against the documentation, annotation syntax is check by the compiler, and possibly also suggested by the IDE.

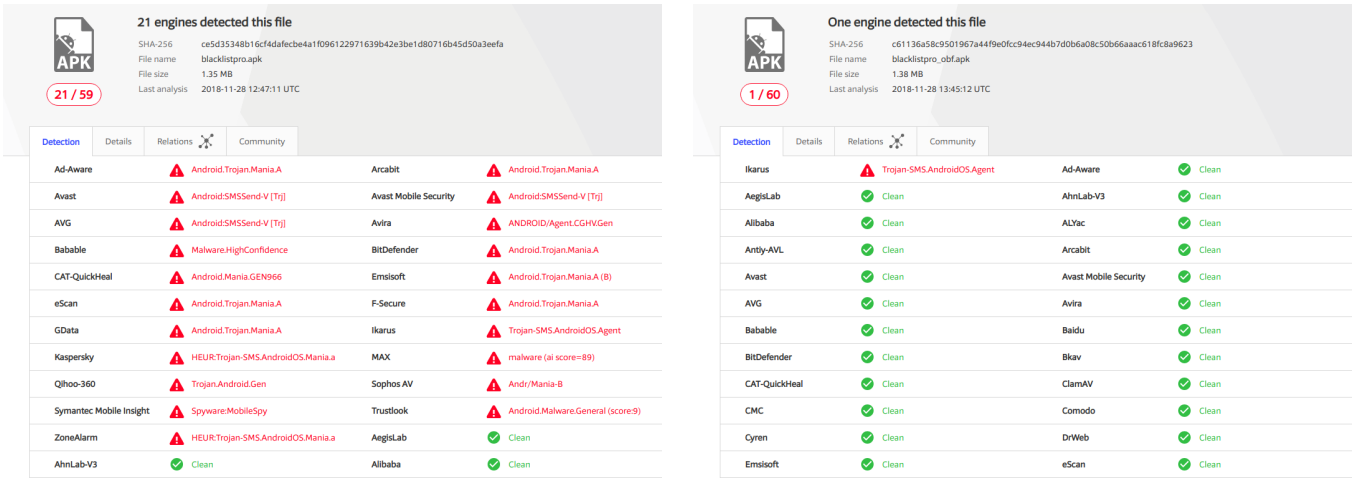
Zelix and *DexGuard*, instead, require a configuration file written in a custom language. Even if configuration templates can be easily found and adapted, developers are supposed to spend some time and familiarize with the new syntax, to be able to customize available configuration templates to their needs. In our case, instead, annotations are used to drive

[7]http://sandmark.cs.arizona.edu/

[8]http://www.allatori.com/

[9]http://www.zelix.com/klassmaster/

[10]https://www.guardsquare.com/en/products/dexguard



(a) Apk without obfuscation (b) Apk with obfuscation

Fig. 4: *Results of the malware detection in our crafted apk with unwanted behaviours*

obfuscations. Annotations are coded in pure Java, which is already familiar to developers.

DexProtector[11] and *DashO*[12] are tools that also provide some support to obfuscation (similarly to *Oblive*), but are more focused on run-time protections, such as anti-tampering and anti-debugging.

VI. Conclusion and Future Work

In this paper we presented *Oblive*, an obfuscation tool implemented as a Gradle plugin, aimed at helping developers in delivering state-of-the-art obfuscation transformations with reduced effort. This plugin uses xor masking with different level of resilience, ranging from low overhead constant masks to advanced techniques requiring the attacker to solve NP-hard problems in order to retrieve statically the obfuscated value. Moreover, *Oblive* uses also an automatic translator from Java code to C code in order to obfuscate entire portions of code while maintaining portability by using cross compilation. We assessed that *Oblive* was able to trick automatic decompilers and evade mainstream malware detection.

As future work, we intend to continue integrating more and more novel obfuscation transformations in *Oblive*, and possibly conduct controlled experiments and usability studies to assess our approach. We also plan to release *Oblive* as open source software.

Acknowledgement

This work has partially been supported by activities “API Assistant/STAnD” and “Teîchos” of the action lines *Digital Infrastructure* and *Digital Finance* of the EIT Digital, and the GAUSS national research project, which has been funded by the MIUR under the PRIN 2015 program (Contract 2015KWREMX).

[11] https://dexprotector.com/

[12] https://www.preemptive.com/products/dasho/overview

References

[1] D. Arp, M. Spreitzenbarth, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. 2014.

[2] M. Ceccato, P. Tonella, C. Basile, B. Coppens, B. D. Sutter, P. Falcarin, and M. Torchiano. How professional hackers understand protected code while performing attack tasks. In *Proceedings of the 25th IEEE International Conference on Program Comprehension (ICPC)*, 2017.

[3] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski. Guest editors’ introduction: Software protection. *IEEE Software*, 28(2):24–27, March 2011.

[4] C.-H. A. Hsieh, J. C. Gyllenhaal, and W.-m. W. Hwu. Java bytecode to native code translation: The caffeine prototype and preliminary results. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 90–99. IEEE Computer Society, 1996.

[5] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[6] J. Nagra and C. Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[7] D. Pizzolotto and M. Ceccato. Obfuscating java programs by translating selected portions of bytecode to native libraries. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 40–49. IEEE, 2018.

[8] R. Tiella and M. Ceccato. Automatic generation of opaque constants based on the k-clique problem for resilient data obfuscation. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 182–192, Feb 2017.

[9] E. Valdez and M. Yung. Software disengineering: Program hiding architecture and experiments. In *International Workshop on Information Hiding*, pages 379–394. Springer, 1999.

[10] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 193–202. IEEE, 2001.

[11] Y. Wang and A. Rountev. Who changed you?: obfuscator identification for android. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pages 154–164. IEEE Press, 2017.