Privacy Preserving Static Analysis

Soham Ghosh
Advisor: Prof. Murali Krishna Ramanathan
Department of Computer Science & Automation
Indian Institute of Science

28^{th} September, 2013

Abstract

It is not always possible for small software development companies to buy highly expensive static tools for debugging their code, but on the other hand open source tools do not perform as good as the paid ones. One way to solve the above problem is by providing static analysis as a service. In this method some third parties provide static analysis as a service instead of selling their tools. The software development companies can easily send their codes to these parties and get their codes analysed and debugged at a very cheap price. But this results in loss of confidentiality. When code is submitted to a third party for finding bugs it is possible that someone with malicious intent may steal and misuse the code.

So we propose Privacy Preserving Static Analysis. This basically means, providing static analysis as a service while not surrendering your code to a stranger. The project suggests a way to achieve this by sending obfuscated code instead of the original code. But in doing so one has to ensure that all bugs in the original code can be mapped to those found in the analysis of the obfuscated version. That is to say that no new bugs are introduced and also no existing bugs are removed as result of the obfuscation. To ensure this certain standard obfuscation techniques have to be relaxed and some techniques need to be more rigid as may be required for a particular bug in consideration. But in doing so raises an important question that - Are we weakening our obfuscation? So the other part of the project deals with measurement of obfuscation. Here we define a novel method to measure the strength of an obfuscated code and also to compare two obfuscated versions of the same code, based on program dependency graphs. Finding out the steps involved in transforming one dependence graph to another gives an estimate of the cost of reverse engineering the obfuscated code and in turn gives a measure for the strength of obfuscation. This process of transformation between graphs can be mapped to a very well known graph theory problem known as Graph Minor.

We are implementing all these in a tool based on an existing obfuscator known as Proguard. We are using FindBugs for static analysis purposes. For evaluation, we are using three large benchmarks namely: Eclipse, Apache Tomcat, Apache Lucene. By comparing the defects found in the original and the obfuscated version of these programs, we found that 12 distinct bugs were introduced after obfuscation and 11 distinct bugs went missing after obfuscation.

1 Introduction

Static analysis, also known as static code analysis, is a method of computer program debugging that is done by examining the code without executing the program. Static analysis finds several applications, in compilers, in tools that help programmers understand and modify programs, and in tools that help programmers verify that programs satisfies certain properties of interest. As software systems and codebases have become larger and more complex there has been a lot of practical interest in using static analysis to assist in detecting software bugs. Over the past few years several static analysis tools have been developed for finding bugs. The most popular among these are: Coverity SAVE [17], Klocwork Insight [18], Parasoft [24] etc. But all of these tools are very expensive, as such it is not always possible for small companies and startups to buy these tools. One way to avoid this is by using open source analysis tools like: FindBugs [4], Saturn [31], Chess [23] etc. But in general it is observed that open source tools do not perform as good as the paid tools. So in exchange of using open source tool these companies have to compromise their software quality and productivity. Apart from static analysis tools being expensive, setting up a static analysis tool

within a company needs special expertise, resources and changes to the build process [22, 3].

One way to solve these above problems is by providing static analysis as a service. In this method some third parties provide static analysis as a service instead of selling their tools. The software development companies can easily send their codes to these parties and get their codes analysed and debugged. The cost of this service is evidently much cheaper than buying a static analysis tool. But there is a crucial drawback in this system, which is loss of confidentiality. It is possible that a malicious resource within these third parties with ill intent can get hold of the original code and misuse it or even sell it for his own gain. So we propose *Privacy Preserving Static Analysis*.

"*The concept of privacy preserving static analysis is that it ensures the same results of applying static analysis on the original code without revealing the original code to the static analyser. The key idea behind this is, that it has to somehow modify the code before sending it to the static analysis service provider such that it is very hard to reverse engineer the modified code.*"

One way to tackle this problem is Code Obfuscation. Obfuscation is a technique used to transform source code so that it is increasingly difficult to understand and harder to reverse engineer. It is a popular technique in the industry as it is used to protect source binaries before final deployment. There has been a significant amount of work in the field of obfuscation. Obfuscation techniques are of several types based on: control flow obfuscation [14], manufacturing opaque predicates [5], design obfuscation [28] etc. Collberg [5] and Batchelder [2] suggest obfuscatation techniques specific to java bytecode. There are quite a few Obfuscation tools which are available in market like: Proguard [19], Zelix KlassMaster [32], JBCO [2]etc.

Using the code obfuscation to protect the original source code, we describe the proposed system of privacy preserving static analysis. The software developer obfuscates his code and sends it to the static analysis service provider. During the obfuscation procedure the software developer maintains a map between the original code and the obfuscated code, which will be used later on for remapping. The static analyser on receiving the obfuscated code performs static analysis on it and sends back a summary of bugs found in the obfuscated code back to the software developer. The software developer, using the map it maintained, maps back the bugs in the summary to the respective bugs in the original code. Thus the developer gets his code analyzed for defects without any risk to confidentiality breach.

But unfortunately the above process does not work so simply. This is because code obfuscatation does not preserve bugs. By running static analysis on several benchmark programs and their obfuscated versions and then comparing the defects found in the original and obfuscated versions, we have seen that the list of defects found do not match. Obfuscation leads to introduction of several new bugs which were not present in the original code. But it is even more interesting to observe that obfuscation also results in loss of defects which were present in the original code. So we need to modify various existing obfuscation techniques in order to ensure preservation of bugs.

The fixes that need to be applied to the obfuscation techniques depend upon the requirements of various bug checkers. Based on these requirements we have to sometimes relax our obfuscatation techniques and then again we may even have to make the obfuscatation techniques more complex. But doing all these raises an important question: How are we affecting the overall obfuscation of the source program? It is possible that the fixes may weaken the entire obfuscatation of the software or it may even strengthen it. So in order to guarantee that our proposed approach does not weaken the obfuscatation, we need some metric to compare the obfuscation strength of two obfuscated versions of the same code. Collberg [5] measures potent and resilient opaque predicates but in a qualitative way. Some other measures of program obfuscatation can relate to instruction count, data-flow, control-flow as distinguished in [1] or it may be cyclomatic complexity as mentioned in [21]. Most of the prior works measure code obfuscatation qualitatively rather than quantitatively. A very recent work [15] suggests on measuring the entropy of an obfuscated code, but unfortunately it only deals with control flow obfuscatations. So even such a metric is not enough to compare two obfuscated versions of the same code based on different obfuscatation techniques.

To address this problem, we propose a metric to measure code obfuscatation and also to compare obfuscated versions of same code irrespective of obfuscation techniques used. Using any program we can construct a program dependence graph, which shows both data dependencies and control dependencies. For e.g., suppose we have a program P and its corresponding program dependence graph is G. Now the program P is obfuscated to give a program P' and let its corresponding program dependence graph be G'. Then transforming graph G' back to G can be thought of as reverse engineering obfuscated program P' to P.

This problem of transforming one graph to another using some specific set of operations can be mapped to very a popular problem in the field of graph theory known as Graph Minor [20]. Although this is a NP-Complete problem, but there exist some approximation algorithms such as [8]. Using these approximation algorithms we can compute the approximate cost of transforming one program dependence graph to its minor. This cost can be assosciated to the cost of reverse engineering and in turn can be used as a metric for code obfuscatation.

As a part of this project we are developing a tool to implement privacy preserving static analysis. The tool will contain an obfuscator specifically modified to preserve bugs. This requires analysing newly introduced and missing bugs to find out the cause of their presence or absence and finally coming up with fixes in the obfuscator to solve this problem. The tool will also contain an implementation of the proposed method to calculate and compare the strength of obfuscated code.

To evaluate privacy preserving static analysis we are using three benchmark programs namely: Eclipse [13], Apache Tomcat [12], Apache Lucene [11]. For obfuscation purposes we are using Proguard [19], on which we are applying our fixes as required by the bug checkers. We are using FindBugs [4] as the static analysis tool for detecting bugs and defects. On comparing the defects found in the original and the obfuscated versions of the three benchmark programs, we found 12 distinct bugs which are introduced after obfuscation and 11 distinct bugs which went missing after obfuscation.

The project has the following technical contributions:

- The project is the first of its kind to define static analysis as a service to avoid the cost of buying highly expensive analysis tools. But at the same time it also preserves the privacy of the analysed programs.
- We also propose a measure for calculating and comparing the strength of obfuscated codes, which gives the true cost of reverse engineering.
- We are implementing all these as a part of a tool which takes a program as input and outputs the obfuscated version of it such that, all defects in the obfuscated code can be mapped to the original program. The tool will also report a value indicating the strength of the obfuscated code.
- We will demonstrate the effectiveness of the tool by running it on the three large benchmark programs.

The rest of the report is structured as follows. In the next section we provide the motivation for the problem along with some examples. Section 3 provides a design overview of privacy preserving static analysis. In section 4 we explain the proposed measure for code obfuscation in details. Finally, in section 5 we present related work and conclude in section 6.

2 Motivation

In this section using examples we show that preserving bugs in obfuscated code is not a trivial issue. We will also show some examples of defects that were introduced or those that went missing, and the approach to fix them.

TYPE	BUG INSTANCE	QUANTITY	REASON
missing	Method names should start with a lower case letter	18	obfuscating with random strings might replace initial uppercase letter of a method with an lowercase character
missing	Field is a mutable array	1	any method with access to the final array field was pulled out of the class and hence the error does not show anymore
missing	Unsynchronized get method, synchronized set method	2	getMethod and setMethod get different obfuscated suffix as a result the checker can't compare
missing	Should be a static inner class	36	inner classes are pulled out during obfuscation
missing	Ambigous invocation of either an inherited or outer method	2	inner classes are pulled out during obfuscation
missing	Class names shouldn't shadow simple name of implemented interface	2	classes and interfaces having the same name may get mapped to different random strings
missing	Class is not derived from an Exception, even though it is named as such	2	classes having exception in their names now get mapped to some random strings
missing	Class defines field that masks a superclass field	1	fields which had same name as another field in a supperclass now get mapped to some random strings
Total		64	

Table 1: Bugs that went missing after obfuscation on benchmark programs `Eclipse`, `Tomcat`, `Lucene`

On comparing the bug summaries for the original and obfuscated versions of the benchmarks we found that several bugs were introduced and several also went missing. Table 1 shows the list of bugs that went missing after obfuscation and their respective reasons for being so. Whereas Table 2 shows the list of bugs that were introduced after obfuscation and their respective reasons for being so. In Table 2 two entries are marked with *, these are the Redundant Nullcheck and Method might ignore exception bugs. The exact

TYPE	BUG INSTANCE	QUANTITY	REASON
extra	non-transient non-serializable instance field in serializable class	8	inner class or even constructors implementing serializable are pulled out as result an object of outer class is created which is non-serializable
extra	Class is serializable, but doesn't define serialVersionUID	14	in general inner classes are not serializable. But when pulled out they need serialID
*extra	Redundant nullcheck of value known to be non-null	2	Dataflow analysis can not decide whether the returned value is null or not in the original code but can determine non-null in obfuscated code
extra	Class names should start with an upper case letter	8	class names are obfuscated with random strings which may contain a lowercase initial letter
extra	Field names should start with a lower case letter	2	field names are obfuscated with random strings which may contain a uppercase initial letter
extra	Unread public/protected field	1	a method which accessed that field might have been moved to another class
*extra	Method might ignore exception	1	Dataflow analysis can not decide whether an exception may occur or not in the original code but can determine a exception situation in obfuscated code
Total		36	

Table 2: Bugs that were introduced after obfuscation on benchmark programs `Eclipse`, `Tomcat`, `Lucene`

cause for the introduction of these two bugs is still to be determined. Although it seems strange, but obfuscatation might have somehow reduced the code complexity as a result of which the dataflow analysis can determine a non-null value or an exception in the obfuscated code but not in the original one.

```
public class OuterClass {
        public static final String name="HELLO";
        private class Inner{
                private Inner(){
                }
                public void print(){
                        System.out.println(name);
                }
                Inner(Inner i){
                        this();
                }
        }
}
```

Listing 1: Original Program containing the inner class within the outer class

```
public class OslkrCakrr
{
  public static final String nkck = "HELLO";
}

class Iiikr
  {
    private Iiikr()
    {
    }

    public void prmia()
    {
      System.out.println("HELLO");
    }
    Iiikr(Iiikr i) {
      OslkrCakrr();
    }
  }
```

Listing 2: Obfuscated Program with the inner class moved out

Now let us look at an example of a missing bug. Listing 1 and Listing 2 shows an example of `Should be a static inner class` bug. In Listing 2 the innerclass is moved outside as result, the static analysis tool identifies it as a normal class and fails to detect the `Should be a static inner class` bug.

Similarly let us look at an example of an extra bug. Listing 3 and Listing 4 shows an example of `Class is serializable, but doesnt define serialVersionUID` bug. In Listing 4 the innerclass is moved outside while making it implement the same Serializable class as it outerclass. But the obfuscator fails to create a serialID for the innerclass. As a result the static analysis tool identifies Listing 4 as a normal Serializable class without any serialID defined. Hence it reports a `Class is serializable, but doesnt define serialVersionUID` bug.

```
public class OuterClass implements Serializable{
        private static final long
        serialVersionUID = 1905162041950251407L;

        private class Inner{
                private Inner(){
                }

                Inner(Inner i){
                        this();
                }
        }
}
```

Listing 3: Original Program with the outer class being serializable

```
public class OslkrCakrr implements Serializable
{
        private static final long
        serialVersionUID = 1905162041950251407L;
}

class Iiikr implements Serializable
  {
    private Iiikr()
    {
    }
    Iiikr(Iiikr i) {
      OslkrCakrr();
    }
  }
```

Listing 4: Obfuscated Program with the inner class being pulled out without having a serialID

A simple solution for the extra bugs problem in Listing 3 and Listing 4 is to make the inner class non-serializable while moving out. But this is not always possible as the innerclass may access some serial fields of the outerclass. A possible solution that is common to both the problems of missing and extra bugs is as follows. The obfuscated code for both the programs in Listing 1 and Listing 3 should have structure as shown in Listing 5. Here the entire content(including the innerclass Inner) inside the original OuterClass is moved out to a newly created class that is, OuterClass1. Basically OuterClass1 is a copy of the old OuterClass. The obfuscated OuterClass now contains only an object of the OuterClass1, using which it can access all its old data members and methods. Moreover the creation of a new class and increasing the level of access reference actually increases the strength of obfuscatation. Hence the proposed fix works quite well for the above two problems.

```
public class OuterClass implements A{
        OuterClass1 ot=new OuterClass1();
}

public class OuterClass1 {
        public static final String name="HELLO";

        private class Inner{
                private Inner(){
                }
                public void print(){
                        System.out.println(name);
                }
                Inner(Inner i){
                        this();
                }
        }
}
```

Listing 5: Solution to the Obfuscation problem

The examples shown in this section is only brief glimpse of the problems that are encountered while running static analysis on obfuscated programs. The examples also motivate to find fixes in obfuscation technique specific to each distinct type of bugs. Keeping in mind the large number of possible bugs we acknowledge that this requires a lot of effort in surveying bugs and coming up with fixes in obfuscatation for the respective bugs.

3 Architecture Overview

The overall system architecture of Privacy Preserving Static Analysis is shown in Figure 1. We broadly divide the system into three major components: `Bug Preserving Obfuscator`, `Static Analysis Service Provider` and `Remapper`.

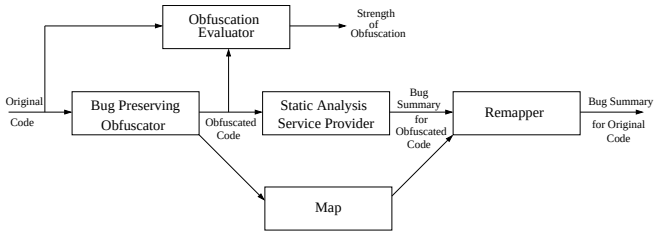

Figure 1: Architecture of Privacy Preserving Static Analysis System

`Bug Preserving Obfuscator` takes as input the original unobfuscated source code and applies certain obfuscation techniques to produce an obfuscated version of the source code, which ensures that no new bugs are introduced and none of the old bugs are missing in the obfuscated code. During this obfuscation process our tool also produces a `Map` which maintains a mapping of obfuscated package, class, method and variable names to their original ones. The map also stores the full classpath for each and every item in case of repackaging of classes or inner classes being moved out of their outer classes. For our tool we use `Proguard Obfuscator` [19] as the basis for all obfuscation techniques applied. We modify the existing obfuscatation techniques within Proguard to guarantee the preservation of bugs.

The obfuscated code is then sent to the `Static Analysis Service Provider`. The analysis provider runs several static analysis techniques on the obfuscated code and reports back a list of found bugs and defects. At this stage even if someone tries to decompile the received code, it is very hard for him to deobfuscate the obfuscated code. For the purpose of our tool and testing we have used `FindBugs` [4] as a static analyser. It reports the discovered bugs in a xml format.

The bug summary for the obfuscated code is then sent back to the sender, where it is passed through the `Remapper`. The `Remapper` remaps the bugs in the obfuscated code to the respective bugs in the original code. It does so by using the contents of the `Map` produced during the obfuscation phase, and then replaces each obfuscated name, line number and classpath with their corresponding original ones. In the end it produces the bug summary for the original code. In our tool the remapper takes FindBugs obfuscated report in xml format and outputs the report for original code also in xml format.

Another essential part of the tool is the `Obfuscation Evaluator`. It takes as input both the original and obfuscated code and then measures the strength of obfuscation. The tool creates the program dependency graph for both the original and obfuscated program, say G and G' respectively. Now using approximation algorithms for solving `Graph Minor` [20] problem, it calculates the cost of transforming G' to G. This cost gives an estimate of the cost of reverse engineering, and in turn gives a measure for the strength of code obfuscation. We explain `Obfuscation Evaluator` in detail in the next subsection.

3.1 Measurement of Obfuscation

In this section we are going to explain in detail the proposed method for measuring the strength of obfuscation including the logic behind it, using examples and step by step procedure for calculating the strength of obfuscatation. We can represent any program using a `Program Dependence Graph`, which represents both control and data dependencies. Any obfuscation technique increases the complexity level of control flow and data dependencies, and any reverse engineering process tries to simplify them. Therefore it seems very natural to argue about the strength of obfuscation using program dependence graph. So in terms of

dependence graphs the process of reverse engineering can be seen as a transformation of the obfuscated dependence graph to the original dependence graph. We will show this using the following example.

```
int square(int x){
        int n;
        n = x * x;
        return n;
 }
```

Listing 6: Original Code for squaring

```
int strvup(int m){
        int o;
        int i=0;
        int s=0;
        while(i < m){
                s = s + m;
                i = i + 1;
        }
        o = s;
        return s;
}
```

Listing 7: Obfuscated Code for squaring

Listing 6 represents a code to calculate the square of a number and Listing 7 represents the obfuscated version of it. The corresponding program dependency graphs are shown in Figure 2 and Figure 3.

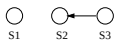

Figure 2: Program dependence graph for Listing 6

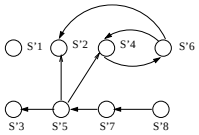

Figure 3: Program dependence graph for Listing 7

Transforming the graph in Figure 3 to that in Figure 2 requires a series of operations such as: `edge and node deletion`, and `edge contraction`. The sequence of steps needed are shown in Figures 4 to 9. From Figures 4 to 9 we see that 5 edge contractions and 1 edge and node deletion is required. During the process of reverse engineering we see that merging of two statements is more costlier than deletion of a statement. Hence we can assign higher weightage to edge contraction operations in comparison to edge and node deletion operations. We assign each edge

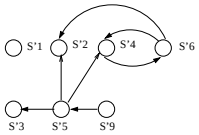

Figure 4: Step 1

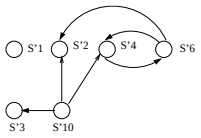

Figure 5: Step 2

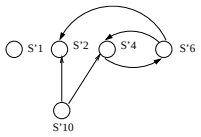

Figure 6: Step 3

contraction operation a weight of 2 and each edge and node deletion operation a weight of 1. Using these weights the cost of reverse engineering or analogously the strength of obfuscatation is: $5*2+1*1 = 11$. This

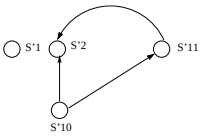

Figure 7: Step 4

method of transforming one graph to another can be mapped to a very popular problem in Graph Theory, known as Graph Minor [20]. Dtermining the steps required for the transformation to find the Graph Minor is a NP-Complete problem. But there exist approximation algorithms [8] for this, using which we can get the approximate steps required. Once we have knowledge of these steps, we can then calculate the strength of obfuscatation using the above weights for any pair of original and obfuscated code.

4 Related Works

The concept of Privacy Preserving Static Analysis and the idea of providing static analysis is completely new and as such there has been no prior work in this field. The work closest to our own is CryptDB project [25], which is in the field of database systems. CryptDB uses homomorphic encryption to run queries securely on relational databases. It encrypts the data in all possible encryption schemes, layered on top of each other. A trusted proxy stands between clients and database system, analyses the SQL queries on the fly, and decrypts the relevant data columns to the right encryption layers so that the query can be executed. In the

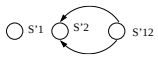

Figure 8: Step 5

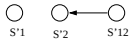

Figure 9: Step 6

field of program analysis, MrCrypt [26] uses the similar concept of homomorphic encryption to securely execute programs on cloud servers without revealing input data. MrCrypt performs type inferencing on a program to identify the set of operations on each input data column, in order to select an appropriate homomorphic encryption scheme for that column. It then encrypts the data column and transforms the program to operate over encrypted data. The encrypted data and transformed program are uploaded to the server and executed as usual, and the encrypted result of the computation is decrypted on the client side. Both of these approaches concentrate on encrypting program data rather than hiding the program structure itself. Moreover the idea of encryption can not be applied for securing program structures as it is not possible to run static analysis on encrypted program semantics. So in our proposed method of Privacy Preserving Static Analysis, obfuscation of code successfuly hides program structure without caring about program data, while keeping the program semantics unchanged.

There has been substantial work in using static analysis to find bugs in programs. Dillig et al. [9] propose a precise technique for a path-sensitive and context-sensitive program analysis. Das et al. [6] present an approach for partial program verification in polynomial time. Engler et al. [10] propose an approach for writing system-specific compiler extensions that automatically checks the code for rule violations. This is in contrast to writing abstract specifications that are then verified by model checkers like SPIN [16] or theorem provers like Z3 [7]. Today several tools exist in the market which provide static analysis to detect bugs [17, 18, 24, 4, 23, 31]. Coverity [3] was the first to commercialize a static analysis tool for bug detection. FindBugs [4] uses bug pattern detectors to find real bugs in several real world Java applications and libraries. Our tool works considering FindBugs as the static analysis tool. But any other bug detecting tool would have worked fine, since in our tool we modified the obfuscatation techniques based on certain properties of bug checkers which are same for most other static analysis tools.

Even in the field of Code Obfuscation there has been quite a lot prior works. A theoretical approach to obfuscations was presented by Collberg et al. [5]. They introduce the concepts of lexical obfuscations (name changing) and data transformations (e.g., splitting boolean values into two discrete numerics that are combined only at evaluation time). However, their chief contributions are in control-flow obfuscations. They make use of opaque predicates to introduce dead code, specifically engineering the dead branches to have buggy versions of the live branches. A technique for combining the data of a program with its control-flow was developed by Wang et al. [30], whereby control and data flow analysis become codependent. Sakabe et al. [27] concentrate their efforts on the object-oriented nature of Java the high-level information in a program. Using polymorphism, they invent a unique return type class which encapsulates all return types and then modify every method to return an object of this type. Unfortunately, their empirical results show significantly slower execution speeds an average slowdown of 30% and a 300% blowup in class file size. Sonsonkin et al. [29] attempt to confuse program structure by suggesting the coalescing of multiple class files into one. For our tool we are building on top of Proguard Obfuscator tool [19].

Although there have been several works related to code obfuscation but unfortunately there are no sufficient or satisfactory work for measuring obfuscation strength. Collberg et al. [5] measures potent and resilient opaque predicates but in a qualitative way. Some other measures of program obfuscatation can relate to instruction count, data-flow, control-flow as distinguished in [1] or it may be cyclomatic complexity as mentioned in [21]. Most of the prior works measure code obfuscatation qualitatively rather than quantitatively. A very recent work [15] suggests measuring the entropy of an obfuscated code, but unfortunately it only deals with control flow obfuscatations. We deal with these problems in our proposed method by representing programs as program dependence graphs, and then draw an analogy between reverse engineering and transformation of obfuscated dependence graph to original dependence graph. Although this is a NP-Complete problem, but there exist some approximation algorithms such as [8]. Using these approximation algorithms we can compute the approximate cost of reverse engineering.

5 Conclusions

We present the notion of Privacy Preserving Static Analysis System and the design for its implementation. The project is the first of its kind to define static analysis as a service, but at the same time it also preserves the privacy of the analysed programs. We also propose a measure for calculating and comparing the strength of obfuscated codes, which gives the true cost of reverse engineering. We will implement all these as a part of a tool which takes a program as input and outputs the obfuscated version of it such that, it preserves all the defects. The tool will also report a value indicating the strength of the obfuscated code. The effectiveness of the tool will be verified by running it on the three large benchmark programs.

References

[1] Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. Program obfuscation: a quantitative approach. In *Proceedings of the 2007 ACM workshop on Quality of protection*, QoP '07, pages 15–20, New York, NY, USA, 2007. ACM.

[2] Micheal Batchelder and Laurie Hendren. Obfuscating java: the most pain for the least gain. In *http://www.sable.mcgill.ca/publications/techreports/sable-tr-2006-5.pdf*.

[3] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

[4] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. Improving your software using static analysis to find bugs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 673–674, New York, NY, USA, 2006. ACM.

[5] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

[6] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 57–68, New York, NY, USA, 2002. ACM.

[7] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Erik D. Demaine, Mohammadtaghi Hajiaghayi, and Ken ichi Kawarabayashi. Algorithmic graph minor theory: Decomposition, approximation, and coloring. In *In 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 637–646. Press, 2005.

[9] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 270–280, New York, NY, USA, 2008. ACM.

[10] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 1–1, Berkeley, CA, USA, 2000. USENIX Association.

[11] Apache Software Foundation. Lucene. In *http://lucene.apache.org/core*.

[12] Apache Software Foundation. Tomcat. In *http://tomcat.apache.org/*.

[13] Eclipse Foundation. Eclipse ide. In *http://git.eclipse.org/c/*.

[14] Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. Control flow based obfuscation. In *Proceedings of the 5th ACM workshop on Digital rights management*, DRM '05, pages 83–92, New York, NY, USA, 2005. ACM.

[15] Roberto Giacobazzi and Andrea Toppan. On entropy measures for code obfuscation. In *ACM SIGPLAN Software Security and Protection Workshop*, 2013.

[16] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[17] Coverity Inc. Coverity save: Coverity static analysis verification engine. In *http://www.coverity.com/*.

[18] Klockwork. Klockwork insight. In *http://www.klocwork.com/*.

[19] Eric Lafortune. Proguard. In *http://proguard.sourceforge.net/*.

[20] Lszl Lovsz. Graph minor theory, 2005.

[21] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[22] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 554–564, New York, NY, USA, 2013. ACM.

[23] Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. Chess: A systematic testing tool for concurrent software, 2007.

[24] Parasoft. Parasoft dottest. In *http://www.parasoft.com/*.

[25] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

[26] Rupak Majumdar Sai Deep Tetali, Mohsen Lesani and Todd Millstein. Mrcrypt: Static analysis for secure cloud computations. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2013, 2013.

[27] Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. Java obfuscation with a theoretical basis for building secure mobile agents. In *In Communications and Multimedia Security*, pages 89–103, 2003.

[28] Mikhail Sosonkin, Gleb Naumovich, and Nasir Memon. Obfuscation of design intent in object-oriented applications. In *Proceedings of the 3rd ACM workshop on Digital rights management*, DRM '03, pages 142–153, New York, NY, USA, 2003. ACM.

[29] Mikhail Sosonkin, Gleb Naumovich, and Nasir Memon. Obfuscation of design intent in object-oriented applications. In *Proceedings of the 3rd ACM workshop on Digital rights management*, DRM '03, pages 142–153, New York, NY, USA, 2003. ACM.

[30] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, Charlottesville, VA, USA, 2000.

[31] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[32] Zelix. Klassmaster. In *http://www.zelix.com/klassmaster/*.