Privacy Preserving Static Analysis

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

28th September, 2013

Abstract

Commercial static analysis tools for software bug detection can discover detects precisely but can be expensive monetarily. Open source static analysis tools may not necessarily have the precision or depth of their commercial counterparts. Furthermore, deploying static analysis tools can add an extra level of complexity to the build process. Changing the usage model of static analysis from being a tool to a service can address these issues. In this model, a server hosts the static analysis as a service and clients submit their source code to the server so that defects can be discovered. However, this approach comes at the cost of revealing the source code by the client to the server (a third party).

In this proposal, we define the problem of *privacy preserving static analysis* which when addressed can overcome the hurdle associated with revealing the source code. Instead of analyzing the original source code, the service analyzes a variant of the original code. The first constraint is that the set of defects detected by the analysis on the original source or its variant must be the same. The second constraint is that deconstructing the original source code from its variant needs to be an arduous task. We address these two constraints by using *code obfuscation* as the variant of the original source code. To handle the first constraint, we modify existing obfuscation techniques. In order to show that the modifications do not reduce the level of obfuscation, we propose a new metric for determining the level of obfuscation (ongoing work).

We will implement our proposal by building a system on top of `ProGuard`, an existing code obfuscator, and `FindBugs`, an open source static analysis tool for `Java` programs. We will evaluate our implementation by analyzing three large benchmarks `eclipse`, `tomcat` and `lucene`. We will show that our implementation addresses the problem of privacy preserving stastic analysis.

1 Introduction

Static analysis is used in a number of applications including compilers, tools for program comprehension and refactoring, tools to verify whether program satisfies certain properties of interest, etc. With the increase in size and complexity of codebases of many software systems, static analysis has also been adopted to detect bugs effectively [3]. Infact, there are a number of commercial static analysis tools for software bug detection [18, 19, 24]. These tools are mostly very precise in identifying defects as they are fine tuned based on real-world experiences.

However, the downside of most commercial tools is that they are expensive. Therefore, it is not always feasible for many software developers to get access to these tools. For example, it may not be feasible for a mobile application developer to be able to afford these expensive tools. Interestingly, the number of mobile applications has grown significantly in the recent past and many of them may not be able to make their software more reliable due to the expenses involved. To address this gap, one can potentially use open source static analysis tools [4, 31]. However, these tools are not fine tuned to be precise, nor do they encompass the entire gamut of defects that need to be detected and they do not have the resources to actively support the users of the tools. Therefore, in exchange for using open source tools, software quality is compromised. Furthermore, irrespective of whether the analysis tool is commerical or not, setting them up to be used on a product regularly requires special expertise, resources and changes to the build process [22, 3].

To overcome all these challenges of using a precise and scalable static analysis tool, one can envision the use of static analysis as a service. In this usage model, the vendors (or developers) of static analysis tools for bug detection provide the tools as a service instead of selling them as products. In the presence of a static analysis service, product developers (e.g., mobile app developers) can subscribe to the service where they are charged on the number of uses. Obviously, using the analysis for a few times is relatively inexpensive as opposed to a fixed term license for a product. Furthermore, users of the service need not worry about changing their internal build or using other local resources to run the static analysis. While this usage model seems appealing at first glance, there is a fundamental challenge that needs to be addressed for it to be practical. Software organizations (or developers) interested in using static analysis as service may not be willing to share their source code with the service provider to ensure protection of trade secrets, copy rights and other legal issues. This results in a *catch-22* situation where the organization cannot share source code with a service provider and the service provider cannot detect defects statically without the presence of source code.

In this proposal, we address the problem of building a practical static analysis system which can be used as a service without the client revealing its source. To the best of our knowledge, we are the first to propose the problem of *privacy preserving static analysis.* We define privacy preserving static analysis as follows:

"*Given source code S', which is a variant of the original source code S, the application of static analysis for software bug detection on S' will produce the same set of defects, D, that is produced by application of the analysis on S, and the effort necessary to obtain S from S' is non-trivial.*"

By this definition of privacy preserving static analysis, if the variant S' is appropriately generated, then S' can be used as the input to the static analysis. The user of the analysis needs to invest non-trivial amount of effort to decipher the original source S from the variant S'. Moreover, from the perspective of the user of the analysis, the set of defects detected is exactly the same without the cost of revealing the source code. Thus the problem of performing privacy preserving static analysis is reduced to the problem of generating a suitable variant of the source code.

While there can be many approaches to generate the variant, we believe code obfuscation [5] fits naturally in this context. Obfuscation is a technique used to transform source code such that reverse engineering the original source from the obfuscated version is hard. This is a popular technique which is widely used [32, 2, 20] before deploying a product. A number of techniques exist for code obfuscation including control flow obfuscation [13], manufacturing opaque predicates [5], design obfuscation [28], etc.

We design a static analysis system where the *client* obfuscates the source code on which defects need to be detected and sends it to the *server* which hosts the static analysis service. The server performs the analysis on the obfuscated code, detects defects and sends the list of defects along with supporting data for triaging the defects to the client. To get the defects on the original source code, the client maintains a map between the original and obfuscated code and uses this map to transform the received defects on obfuscated code into the defects on the original code.

While the above design satisfies the confidentiality of the source code, it does not necessarily satisfy the second property of obtaining the same set of defects. Based on our initial experiments, running static analysis [4] on several large benchmarks [11, 10, 12] and their corresponding obfuscated versions, we observe that the defects detected in the original and the obfuscated versions are not *exactly* the same. Based on the obfuscation and static analysis techniques employed, obfuscation results in the introduction of several new defects being detected which are not present in the original code. More interestingly, obfuscation also causes the static analysis to not detect defects that exist in the original code which would have been detected by the analysis in the absence of obfuscation. To satisfy the second property of obtaining the same set of defects, the underlying obfuscation algorithm may need to be appropriately modified. Although the static analysis can also be modified but that is not desirable.

However, any modification to the underlying obfuscation algorithm raises an important question on whether the design compromises on the first property (requiring significant effort for reverse engineering) to ensure that the second property is satisfied. For example, it is possible that the modifications done to the obfuscation algorithm may make it easier for a malicious server to reverse engineer. To address this concern effectively, our approach needs to provide guarantees that the modifications do not weaken the first property. In that direction, we need to design a suitable metric to measure the level of obfuscation of any source code (ongoing work).

We propose to incorporate the above design by implementing a privacy preserving static analysis tool. The tool will contain a custom-built obfuscator that satisfies the two properties in the definition of privacy preserving static analysis. The tool will also include a client-server implementation to handle the overall analysis. Finally, we propose to have measurement of the level of obfuscation as a feature of the tool. We intend to use `Proguard` [20] as a building block for our custom-built obfuscation and use `findbugs` [4] as the static analysis. To evaluate our design and implementation, we will use three benchmark programs – `eclipse` [12], `apache-tomcat` [11] and `apache-lucene` [10]. We will show that the set of defects detected on the original and obfuscation versions are identical. We will also show that our custom-built obfuscation does not reduce the overall obfuscation. ‘

We intend to make the following technical contributions:

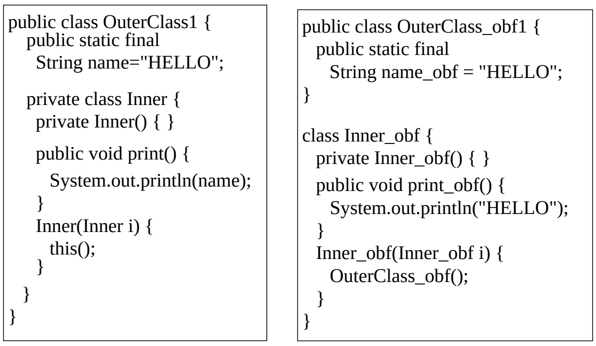
- We are the first to propose the problem of privacy preserving static analysis so that static analysis for software bug detection can be used as a service.
- We design and implement a privacy preserving static analysis tool.
- We propose a novel metric for determining the level of obfuscation – the cost of reverse engineering.
- We will demonstrate the effectiveness of our implementation by analyzing three large benchmarks.

2 Motivation

In this section, we motivate our proposal further by showing that existing obfuscation are not applicable directly. We use examples to show that new defects are detected on the obfuscated version that do not existed in the original source. Similarly, defects that exist in the original source are masked by the obfuscation. We will briefly provide pointers on how to fix the gain (or loss) of defects.

Table 1 shows statistics on the defects that are not detected anymore by `findbugs` on obfuscated versions of the three benchmarks. Table 2 shows the list of defects that are newly detected by `findbugs` due to obfuscation. These defects are not detected on the original code.

Now let us look at an example of a missing defect. Figure 1(a) shows an example of a defect in the code suggesting that the inner class (`Inner`) should be static. But after obfuscation the inner class is moved out of the encompassing (`OuterClass1`) as shown in Figure 1(b) and obfuscated. The result is that the static analysis tool identifies it as a normal class and fails to detect the defect that exists in the original code.



```
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();
     }
   }
}
```

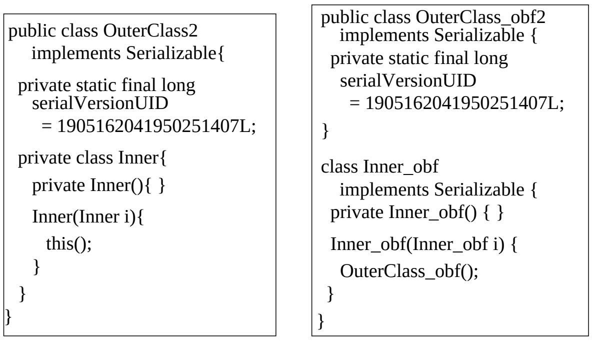
```
public class OuterClass_obf1 {
   public static final
     String name_obf = "HELLO";
}

class Inner_obf {
   private Inner_obf() { }
   public void print_obf() {
     System.out.println("HELLO");
   }
   Inner_obf(Inner_obf i) {
     OuterClass_obf();
   }
}
```

(a) Original Program containing the inner class within the outer class

(b) Obfuscated Program with the inner class moved out

Figure 1: Missing Defects

Similarly, let us look at an example of an extra defect. Figure 2(a) shows an example in which the outer class is serializable and has its `serialVersionUID` defined. After obfuscation as shown in Figure 2(b), the inner class is moved outside while making it implement the same Serializable class as its outer class. But the obfuscator fails to create a `serialVersionUID` for the inner class. This results in the introduction of a new defect suggesting that a class that is serializable does not define a `serialVersionUID`.



```
public class OuterClass2
    implements Serializable{

  private static final long
    serialVersionUID
      = 1905162041950251407L;

  private class Inner{
    private Inner(){ }

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

```
public class OuterClass_obf2
    implements Serializable {
  private static final long
    serialVersionUID
      = 1905162041950251407L;
}

class Inner_obf
    implements Serializable {
  private Inner_obf() { }

  Inner_obf(Inner_obf i) {
    OuterClass_obf();
  }
}
```

(a) Original Program with the outer class being serializable

(b) Obfuscated Program with the inner class being pulled out without having a serialID

Figure 2: Extra Defects

A simple solution for the extra defects problem in Figure 2(a) and Figure 2(b) is to make the inner class non-serializable while moving out. But this is not always possible as the innerclass may access some serial

Checker	Count	Reason for loss
Method names should start with a lower case letter	18	obfuscating with random strings might replace initial uppercase letter of a method with a lowercase character
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
Unsynchronized get method, synchronized set method	2	getMethod and setMethod get different obfuscated suffix as a result the checker can't compare
Should be a static inner class	36	inner classes are pulled out during obfuscation
Ambigous invocation of either an inherited or outer method	2	inner classes are pulled out during obfuscation
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
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
Class defines field that masks a superclass field	1	fields which had same name as another field in a superclass now get mapped to some random strings
Total	64	

Table 1: Statistics of defects not detected by `findbugs` after `eclipse`, `tomcat`, `lucene` are obfuscated.

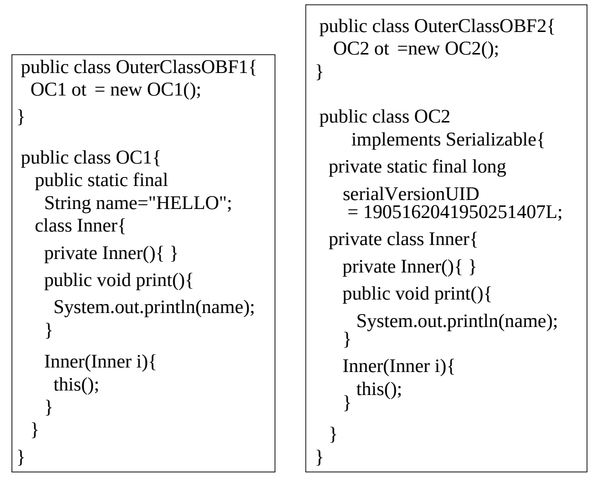
Checker	Count	Reason for gain
non-transient non-serializable instance field in serializable class	8	inner class or even constructors implementing serializable are pulled out as result an object of outer class is created which is non-serializable
Class is serializable, but doesn't define serialVersionUID	14	in general inner classes are not serializable. But when pulled out they need serialID
*Redundant nullcheck of value known to be non-null	2	Dataflow analysis cannot decide whether the returned value is null or not in the original code but can determine non-null in obfuscated code
Class names should start with an upper case letter	8	class names are obfuscated with random strings which may contain a lowercase initial letter
Field names should start with a lower case letter	2	field names are obfuscated with random strings which may contain a uppercase initial letter
Unread public/protected field	1	a method which accessed that field might have been moved to another class
*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: Statistics of defects newly detected by `articlefindbugs` after `eclipse`, `tomcat`, `lucene` are obfuscated. The exact reason for the gain on the entries prefixed with * is to be determined.

fields of the outerclass.

A possible solution to the missing defects problem is to obfuscate the code as given in Figure 3(a). The entire content inside the original encompassing class, `OuterClass1` of Figure 1(a), including the innerclass `Inner` is moved to a newly created class named `OC1`. Basically `OC1` is a copy of the old `OuterClass1` (in Figure 1(a). The obfuscated version of the `OuterClass1` (`OuterClassOBF1`) now contains only an object of `OC1`, using which it can access all its old data members and methods.

Similarly, for the extra defect problem a possible solution is to obfuscate the code as given in Figure 3(b). Like in the previous solution, the entire content inside the original encompassing class `OuterClass2` of Figure 2(a), including the innerclass `Inner` is moved to a



```
public class OuterClassOBF1{
  OC1 ot  = new OC1();
}

public class OC1{
   public static final
     String name="HELLO";
   class Inner{
     private Inner(){ }
     public void print(){
       System.out.println(name);
     }
     Inner(Inner i){
       this();
     }
   }
}
```

(a) Solution to missing defect in Figure 1(b)

```
public class OuterClassOBF2{
  OC2 ot  =new OC2();
}

public class OC2
      implements Serializable{
  private static final long
    serialVersionUID
     = 1905162041950251407L;
  private class Inner{
    private Inner(){ }
    public void print(){
      System.out.println(name);
    }
    Inner(Inner i){
      this();
    }
  }
}
```

(b) Solution to extra defect in Figure 2(b)

Figure 3: Solution addressing the extra and missing defects

newly created class named `OC2`. Basically `OC2` is a copy of the old `OuterClass2` (in Figure 2(a)). The obfuscated version of the `OuterClass1` (`OuterClassOBF2`) now contains only an object of `OC2`, using which it can access all its old data members and methods.

In the above two solutions, all the data accesses and function calls of the old outer class is redirected through an object of this new class. Since the suggested technique is built on top of the existing obfuscation technique, it actually makes the existing obfuscation technique more stringent. We also observe that defects are neither lost nor gained for the Figure1(a) and Figure2(a).

The examples shown in this section provide a brief overview of the problems that are encountered while running static analysis on obfuscated programs. Furthermore it is not clear whether the modified obfuscation actually increases or decreases the level of obfuscation, even though we claim that it has increased. Therefore instead of a subjective argument, an objective metric to determine the level of obfuscation will be helpful. This motivates the design of the obfuscation metric.

3 Architecture Overview

The overall system architecture of Privacy Preserving Static Analysis is shown in Figure 4. We broadly divide the system into three major components: `Obfuscator`, `SA Server` and `Remapper`. An additional unit `Obfuscation Evaluator` measures the strength of obfuscation.

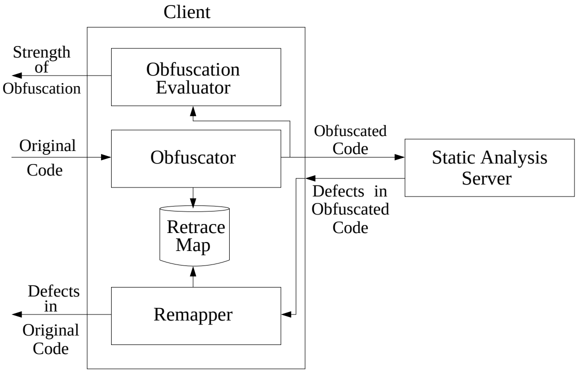


Figure 4: Architecture of the Privacy Preserving Static Analysis System

`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 defects are introduced and none of the old defects are missing in the obfuscated code. During this obfuscation process our tool also produces a `Retrace 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 system we use `Proguard` [20] as the basis for obfuscation. We modify the existing obfuscation techniques in Proguard to guarantee the preservation of defects.

The obfuscated code is then sent to the `SA Server`. The SA Server runs several static analysis techniques on the obfuscated code and reports back a list of detected defects. At this stage even if someone tries to reverse engineer the received code, it will be a nontrivial task. In this project, we are using `FindBugs` [4] to perform the static analysis.

The defect summary for the obfuscated code is then sent back to the client, where it is passed through the `Remapper`. The `Remapper` remaps the defects in the obfuscated code to the respective defects in the original code. It does so by using the contents of the `Retrace Map` produced during the obfuscation phase, and then replaces each obfuscated name, line number and classpath with the original values . In the end it produces the defect summary for the original code.

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 (ongoing work).

4 Related Work

4.1 Preserving Privacy Techniques

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 secured queries on relational databases. It applies the encryption schemes on the data in a sequential manner. The SQL queries are analysed on the fly, and the data columns are decrypted to the required encryption layers to execute the query on the server.

In the 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 applied on each input data column, and appropriately chooses a homomorphic encryption scheme for that column. The data columns are encrypted and the program is also transformed accordingly to operate over the encrypted data. The encrypted data and transformed program are then sent to the server for execution, and the encrypted result of the computation is decrypted on the client side. Both these approaches concentrate on encrypting program data rather than hiding the program structure itself. Moreover the idea of encryption cannot 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.

Applied largely in the field of cryptography, Zero-Knowledge proofs play an important role in privacy preservation. Goldwasser et al. [16] conceived the idea of ZKPs and suggested the concept of knowledge complexity, a measurement of the amount of knowledge about the proof transferred from the prover to the verifier. Goldreich et al. [15] constructs a ZKP for the graph three-colorability problem and then further extends the proof to show that there exists a ZKP for any language in NP. The Fiege-Fiat-Shamir identification scheme [9] is the basis of most zero-knowledge indetification protocols currently in use in the domain of cryptography.

4.2 Static Analysis

There has been substantial work in using static analysis to find bugs in programs. Dillig et al. [7] propose a technique to precisely compute neccessary and sufficient conditions required by program points that are path-sensitive and context-sensitive.Instead of collecting abstract specifications that are then verified by model checkers like SPIN [17] or theorem provers like Z3 [6], Engler et al. [8] propose an approach using metalevel compilation (`MC`) to automatically check system rules. Today several tools exist in the market which provide static analysis to detect bugs [18, 19, 24, 4, 23, 31]. FindBugs [4] uses bug pattern detectors to find real bugs in several real world Java applications and libraries. Our tool uses FindBugs to perform the task of static analysis. But any other tool detecting defects would also work fine, since in our tool we modified the obfuscation techniques based on certain properties of defect checkers, and not the implementation of the analysis itself.

4.3 Obfuscation Techniques

There has been a lot of prior work in the field of code obfuscation. An initial theoretical approach to obfuscation is presented by Collberg et al. [5]. They introduce the concepts of lexical obfuscation (random strings used for naming) and data transformations (such as obtaining a boolean value at evaluation time from the combination of two distinct expressions). However, their chief contributions are in control-flow obfuscation. They exploit the difficulty of analyzing aliases and concurrent programs to construct opaque predicates. Wang et al. [30] suggests combining the data of a program with its control-flow and thereby making control and data flow analysis interdependent. Sakabe et al. [27] concentrate their efforts on the object-oriented nature of Java. Using polymorphism, they invent a universal datatype class to encapsulate all return types and method parameters thus hidding their true types. Sonsonkin et al. [29] attempt to confuse program structure by combining multiple class files into one. For our system we are building on top of Proguard tool [20] and can use any of the suitable obfuscation technique.

4.4 Measure of Obfuscation

Although there have been several approaches related to code obfuscation, we are unaware of any 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 obfuscation relate to instruction count, data-flow, control-flow as discussed in [1] or it may be cyclomatic complexity as mentioned in [21]. Most approaches measure code obfuscation qualitatively rather than quantitatively. A very recent work [14] suggests measuring the entropy of an obfuscated code, but unfortunately it only deals with control flow obfuscation. We will deal with these problems in our project by defining a new metric for measuring the strength of obfuscation (ongoing work).

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] 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.

[7] 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.

[8] 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.

[9] U. Fiege, A. Fiat, and A. Shamir. Zero knowledge proofs of identity. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 210–217, New York, NY, USA, 1987. ACM.

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

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

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

[13] 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.

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

[15] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *J. ACM*, 38(3):690–728, July 1991.

[16] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. ACM.

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

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

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

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

[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/.*