On Evaluating Obfuscatory Strength of Alias-based Transforms using Static Analysis

Anirban Majumdar, Antoine Monsifrot and Clark Thomborson
Secure Systems Group
Department of Computer Science
The University of Auckland
Auckland, New Zealand Private Bag 92019
Email: {anirban, antoine, cthombor}@cs.auckland.ac.nz

***Abstract*— Aliasing occurs when two variables refer to the same memory location. This technique has been exploited for constructing resilient obfuscation transforms in languages that extensively use indirect referencing. The theoretical basis for these transforms is derived from the hard complexity results of precisely determining which set of variables refer to the same memory location at a given program point during execution. However, no method is known for randomly generating hard problem instances. Unless we are able to evaluate the obfuscatory strength of these transforms using static analysis tools, we cannot correlate the resilience expected in theory with what actually holds in practice. In this contribution, we will outline the main difficulties in experimentally evaluating obfuscatory strength and give an overview of techniques that are suited for analysing well-established alias-based obfuscation transforms.**

I. Introduction

Static analysis is complicated in the presence of aliasing. Precise and flow-sensitive alias analysis is undecidable in languages with dynamic allocation, loops, and if-statements [1]. Based on this observation, several obfuscation techniques claim the intractability of precise alias analysis as a theoretical basis for their transforms. Most notable among these are:

- Opaque predicates rely on difficulty of the alias analysis problem and the shortcomings of the available conservative algorithms for analysing the control-flow of programs [1], [2].
- Control-flow flattening techniques which rely on the `NP-hardness` of precisely determining the indirect branch target addresses of dispatchers in the presence of aliased pointers [3].
- Obfuscation techniques like identifier renaming [4], class coalescing and class splitting [5] which rely on the difficulty that in the presence of method-overloading, precisely determining if there exists an execution path in a program for which a given reference points to a given method at a point of the program execution is `NP-complete`[6], [7].

Even though these obfuscation techniques are based on intractable computational problems, we do not know, in practice, how to arbitrarily generate sufficiently hard obfuscated problem instances such that all program analysis techniques would *fail* (i.e. give imprecise, unanalysable results, or be unscalable, run out of memory, crash, or never terminate). Unfortunately, none of the previous authors, while concentrating on the performance impact of obfuscation, provided empirical results of their security analyses. The main hindrance is the inability to answer the following question: *What sort of analysis tools are useful for the automated understanding of the code obfuscated with aliasing transforms?* Available static program analysis tools, mostly academic projects, are developed mainly for program optimisation and not for security analysis. A supplementary question is: *Can general purpose program analysis tools be used to assess the obfuscatory strength of aliasing transforms or do we need to develop customised analysis tools instead?* At the same time, even if we are able to find a suitable tool for one particular technique, how do we tell that the tool has successfully "cracked" the obfuscation technique unless we know what an obfuscation transform is supposed to protect? Can we guarantee that all general tools of that category (and its improved versions) can "crack" any general instance of code obfuscated with this technique?

In this contribution, we will summarise our observations from our experience with available static analysis tools. We will start the next section with an outline of alias-based opaque predicates and their analysis using a pointer analysis framework and a static slicer. We will conclude the paper with a description of an emerging technique called concept analysis, showing how it can be used to analyse the class of obfuscation techniques which draw obfuscatory strength from overloading.

II. Analysing Opaque Predicates

An *opaque predicate* is a conditional expression whose value is known to the obfuscator, but is difficult for an adversary to deduce statically. A predicate Φ is defined to be *opaque* at a certain program point p if its outcome is only known at obfuscation time. Following Collberg et al. [2], we write $\Phi_p^F(\Phi_p^T)$ if predicate Φ always evaluates to False (True) at program point p for all runs of the same program. We call such predicates *Opaquely True (False)* at program point p. The notation $\Phi_p^?$ is used to denote *Opaquely Unknown* predicate,

```
public class Test {
   public static void main(String[] args) {
      Test x = new Test();
      Test a = (Test) id(x);
      Test b = (Test) id(x);
      if(a==b)
         System.out.println("false predicate");
   }
   static Object id(Object o) {
     Test p = new Test();
     return p;
   }
}
```

Fig. 1. Example of an opaque predicate needing allocation site sensitive alias analysis tool to correctly determine its value

i.e. one whose value depends on a program input supplied by the user, by the operating system, or by some program such that it sometimes evaluates to true and sometimes to false during different program executions. The opaqueness of such predicates is necessary for the resilience of all known control-flow transformations. Collberg et al. [1] used the intractability property of pointer aliasing to construct aliased opaque predicates. Their construction is based on the fact that it is impossible for approximate and imprecise static analysers to detect all aliases all of the time.

A. Experiments with an alias analysis framework

We simulated an attack on aliased opaque predicates using BDDBDDB (Binary Decision Diagram-Based Deductive DataBase) [11], an alias analysis tool for Java. The objective was to see what useful information can be obtained by an adversary equipped with state-of-the art alias analysis tool such as BDDBDDB.

BDDBDDB is an implementation of Datalog, a declarative programming language for specifying program analysis. The tool is *context-sensitive*, which means it can distinguish between different calling contexts of a method and thus prevents erroneous propagation of alias information from one caller to another of the same method. It is also very scalable and can scale up to analysis of nearly 700,000 bytecodes. The analysis is also *field sensitive* meaning that it can track individual fields of individual pointers. *Flow sensitivity* in BDDBDDB is intraprocedural. The basic approach is the use of *cloning*. Cloning generates multiple instances of a method such that every distinct calling context invokes a different instance, thus preventing information from one context to flow to another. Thus, through cloning, context-sensitivity can be generated using context-insensitive algorithm by applying it to each of the clones.

BDDBDDB, however, is not *allocation site sensitive*. Objects are referred by their allocation sites and since allocation sites are not context-sensitive, BDDBDDB approximates one allocation site for every calling context of an object. This results in BDDBDDB failing to report that the variables a and b alias in the predicate of the example in Fig. 1. We exploited this analysis weakness by subjecting the program in Fig. 2 to BDDBDDB analysis. Two different analysis techniques were used to analyse this program: one in which only the context sensitivity aspect of aliasing was tested (referred hereafter as CS - meaning only Context Sensitivity) and the other one

```
public class Test {
   static Test Testcomm = new Test();
   Test(){}
   public static void main(String[] args){
      Test x = new Test();
      Test y = new Test();
      Test a = (Test) id(x);
      Test b = (Test) id(x);
      Test c = (Test) id(y);
      Test d = (Test) id1(x);
      Test e = (Test) id1(x);
      Test f = (Test) id2(x);
      Test g = (Test) id2(y);
      Test h = (Test) id3(x);
      Test i = (Test) id3(x);
      Test j = (Test) id3(y);
    }
    static Object id(Object o){return o;}
    static Object id1(Object o){
      Test p = new Test();
      return p;
    }
    static Object id2(Object o){
      Test p = new Test();
      return Testcomm;
    }
    static Object id3(Object o){
      Test p = (Test) id(o);
      return p;
    }
}
```

Fig. 2. Test program used for performing CS and CSAS analyses

	x	y	a	b	c	d	e	f	g	h	i	j
x	T	F	T	T	F	F	F	F	F	T	T	F
y		T	F	F	T	F	F	F	F	F	F	T
a			T	T	F	F	F	F	F	T	T	F
b				T	F	F	F	F	F	T	T	F
c					T	F	F	F	F	F	F	T
d						T	F	F	F	F	F	F
e							T	F	F	F	F	F
f								T	T	F	F	F
g									T	F	F	F
h										T	T	F
i											T	F
j												T

need an allocation site sensitive analysis

not correctly analysed

Fig. 3. Analysis result obtained using BDDBDDB on program of Fig. 2.

in which both allocation site sensitivity context sensitivity tests were performed (referred hereafter as CSAS - meaning Context Sensitivity with Allocation Site sensitivity).

The results are presented in Fig. 3. Each boolean in this table indicates whether the variables in the corresponding row and column alias each other or not. The white cells correspond to correct CS as well as CSAS analyses of aliasing using BDDBDDB. Red cell (darker shade) (variable `d` alising variable `e`) correspond to correct context sensitive (CS) but incorrect allocation site sensitive analysis (CSAS). Finally, the blue cells (lighter shade) correspond to an incorrect CSAS and CS analyses. This could be because of an extra level of indirection incurred through methods `id3` and `id`. A rather interesting observation is that CSAS analysis is true for all the blue cells and whereas CS analysis is false for the same. A plausible explanation could be the tool's inability to clone contexts past one level of indirection (however this cannot be corroborated from [11]).

Here, we come to the conclusion that multilevel aliases

which exploit the allocation site sensitivity problem can potentially confuse an adversary armed with even latest tools like BDDBDDB into giving imprecise results. This is a promising observation for manufacturing aliased opaque predicates of [1]. Later [12] observed that determining whether a predicate is opaquely true or false reduces to solving the "must-alias" problem. BDDBDDB and other alias analysis tools are useful for analysing the "may-alias" problem (which is often required for compiler optimisation) but the alias information provided by such tools is inconclusive for determining whether a predicate will be opaquely true at a particular program point. However, a "may-alias" is sufficient for finding out opaquely false predicates.

B. Experiments with a static slicer

Even if the static determination of the outcome of opaque predicates using an alias analyser fails, significant understanding of how the predicate variables are updated can be obtained if an adversary manages to find out the invariants involved in maintaining this predicate. This attack can be mounted using a static slicer which when given a certain slicing criterion (say the opaque predicate variable) will slice (by marking) relevant parts of the program that updates the predicate [13]. In this part of the experiment, we try to determine how such a slicing attack could be mounted and suggest a possible remedy to deter it.

We selected the Indus [14] Java slicer to simulate this attack. Indus was chosen because it uses Soot [9] to derive points-to results and has reasonable analysis precision [15]. Indus incorporates a program slicing library which facilitates different high level program analyses such as dependences on intra- and inter-procedural data, control, interference, and synchronization of the program. The slicing criterion for Indus could be a source code line or a Jimple [1] statement.

We used the program fragment of Fig. 4 to simulate a slicing attack on opaque predicates using Indus. Here, we used a predicate statement as slicing criteria. A precise slice of all statements affecting the slicing criteria constitutes a successful attack. In our example, we inserted an aliased opaque predicate if(g != h.selectNode(2)) in a fragment from the SciMark benchmark for LU decomposition, a method for decomposing a matrix into a product of lower triangular and upper triangular matrices. A particular method `solve` has been shown which performs solving by substitution. We inserted updates to the dynamic Node data structure in different methods of LU and inserted predicates in `solve`. The methods which update the Node structure has been omitted in the illustration. Indus slices the statements (shown in **highlight**) g.addNode(2) and h.addNode(1) in the method solve. A simple slicing attack could be mounted with the attacker slicing a program based on an opaque predicate as the slicing criterion. If the slice is input invariant [2], the attacker can safely remove the statements in the slice and replace the predicate with a constant without compromising the correctness of the program.

[1]Jimple is an intermediate representation used in the Soot framework

[2]An input invariant slice does not depend on keyboard input or file read

```
public static void solve (double LU[][], int pvt[], double b[]) {
    int M = LU.length;
    int N = LU[0].length;
    int ii = 0;

  g.addNode(2);

    for (int i=0; i<M; i++)
    {
        int ip = pvt[i];
        double sum = b[ip];
        b[ip] = b[i];
        if (ii==0) {
            for (int j=ii; j<i; j++)
                sum -= LU[i][j] * b[j];
        { else {
            if (sum == 0.0)
                ii = i;
        }
        b[i] = sum;
    }

  h.addNode(1);

    for (int i=N-1; i>=0; i--)
    {
        double sum = b[i];
        for (int j=i+1; j<N; j++)
            sum -= LU[i][j] * b[j];
        b[i] = sum / LU[i][i];
    }

  if (g !=h.selectNode(2))

        b[1] = b[1] + 1;
}
```

Fig. 4. Example of a program slice using Indus. The slicing criterion selected here is the opaque predicate if(g != h.selectNode(2)). Statements in the slice are labeled with **highlight**.

Collberg and al. [2] proposed a solution for increasing the stealth of opaque predicates by merging classes used in building the obfuscating data structure with the other similar user-defined class. We argue here that this modification will still not resist slicing attacks since slicing is done at the statement level and is not affected by class hierarchy of the program.

Our solution is illustrated in Fig. 5. Here, the invariant maintaining code is interwoven with the original program code. The predicate maintaining data structure code is correlated with original program data structure by introducing a bogus conditional, which cannot affect our opaque predicate, such as if(b[1] $>$ b[2]) h.addnode(1). Using the same slicing criteria as in Fig. 4, we get a much bigger slice for the program in Fig. 5. Thus, by introducing such simple correlations, we can make it difficult for the adversary to blindly strip off the sliced code from obfuscated program.

III. Analysing identifier renaming obfuscation

Tyma's patent [4] and his *Dotfuscator* [19] and *DashO* [20] products, for obfuscating Microsoft Intermediate Language (MSIL) and Java Bytecode respectively, transform identifiers of multiple methods into an identifier of smaller length. For unrelated methods, identifier renaming gives an effect of "overload induction"; resolving the actual methods that are to be invoked at runtime reduces to the hardness of pointer aliasing problem. Fig. 6 shows a simple program which has been obfuscated using Tyma's identifier renaming algorithm to one in Fig. 8 (adapted from [18]).

In this part of the experiment, we inspect whether an attacker armed with a concept lattice visualiser can extract meaningful information out of the obfuscated code. Concept

```
public static void solve (double LU[][], int pvt[], double b[]) {
    int M = LU.length;
    int N = LU[0].length;
    int ii = 0;
    g.addNode(2);
    for (int i=0; i<M; i++)
    {
        int ip = pvt[i];
        double sum = b[ip];
        b[ip] = b[i];
        if (ii==0) {
            for (int j=ii; j<i; j++)
                sum -= LU[i][j] * b[j]; {
        else {
            if (sum == 0.0)
                ii = i;
        }
        b[i] = sum;
    }
    if (b[1] > b[2])
        h.addNode(1);
    for (int i=N-1; i>=0; i--)
    {
        double sum = b[i];
        for (int j=i+1; j<N; j++)
            sum -= LU[i][j] * b[j];
        b[i] = sum / LU[i][i];
    }
    if (g !=h.selectNode(2))
        b[1] = b[1] + 1;
}
```

Fig. 5. Example of program slice of the same method of the Fig. 4 after inserting our correlation code if(b[1] $>$ b[2]) h.addnode(1); . The slicing criteria is same as before. Statements in the slice are labeled with **highlight** .

```
public class test1{
    private int term1;
    private int term2;
    private boolean areRelativelyPrime;

    public test1(int term1, int term2){
        this.term1=term1;
        this.term2=term2;
        areRelativelyPrime=areRelativelyPrime();
    }

    public static int gcd(int term1, int term2){
        int remainder;
        remainder=term1%term2;
        if (remainder==0){
            return term2;
        }
        else{
            return gcd(term2, remainder);
        }
    }

    private boolean areRelativelyPrime(){
        if (gcd(term1, term2)==1){
            return true;
        }
        else{
            return false;
        }
    }

    public static void main(String args[]) {
      test1 a=new test1(12, 19);
    }
}
```

Fig. 6. A simple gcd calculating test code.

lattices are natural inheritance structures and a natural application domain is in the understanding of class hierarchies for object-oriented languages. Snelting and Tip [16] proposed the use of concept lattices in software engineering domain for making refactoring decisions. We tested the hierarchies of both the programs with the KABA (Klassen Analyse mit Begriffs Analyse) [17] prototype class refactoring system and

```
public class a{
    private int a;
    private int b;
    private boolean c;

    public a(int a, int b){
        this.a=a;
        this.b=b;
        c=c();
    }

    public static int b(int a, int b){
        int c;
        c=a%b;
        if (c==0){
            return b;
        }
        else{
            return b(b, c);
        }
    }

    private boolean c(){
        if (b(a, b)==1){
            return true;
        }
        else{
            return false;
        }
    }

    public static void main(String args[]) {
      a b=new a(12, 19);
    }
}
```

Fig. 7. Obfuscated code using identifier renaming.

concept lattice visualisation system. In the KABA display editor, every box represents a class, its name is printed in bold font in the centre (nodes containing only members from the same original class C are named $C'n$. Members are displayed above the class name, variables below it. To reduce the screen space requirements, attributes and objects are not displayed by default; scroll arrows next to the class name allow the to expand them if necessary.

Fig. 8 shows the corresponding concept lattice for the code in Fig. 6. `term1` and `term2` have been factored out in `test1'1` and `areRelativelyPrime` in `test1'2`. Because of the use of fields `term1` and `term2` in `test1()` and field `areRelativelyPrime` in `areRelativelyPrime()`, both methods have been defined in a lower hierarchical order at `test1'3`.

Fig. 9 shows the concept lattice for the obfuscated code in Fig. 7. It has the same hierarchical structure and factors out variables and methods in the same manner as for code in Fig. 6. KABA's refactoring algorithm (based on *client usage*) was applied to both the concept lattices and their similar simplified lattice structures are shown in Fig. 10 and 11 respectively.

Our simple attack with KABA was not completely successful at deobfuscating our sample obfuscation using Tyma's patented method. If we consider the relaxed goal of obfuscation, i.e. the objective is to make the obfuscated code *unintelligible*, then we can say that Tyma's method *fails* in this instance, as the obfuscated code reveals the same amount of information as its unobfuscated counterpart when analysed using the same static analysis tool. That is, if the code of Fig. 6 can be attacked by adversaries, then its obfuscated counterpart would have a similar vulnerability. The obfuscated code might "look" obscure because of overusing identifiers but it is equally analysable as the unobfuscated code. We note that the built-in alias analysis engine of KABA was able

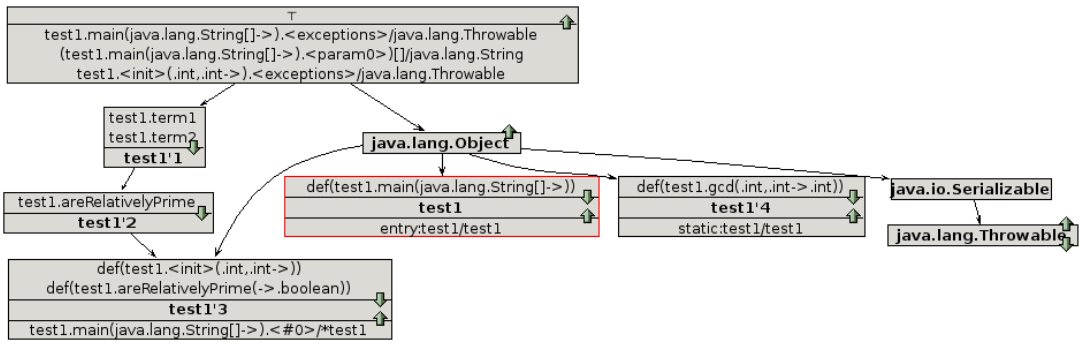

Fig. 8. Concept lattice for program in Fig. 6

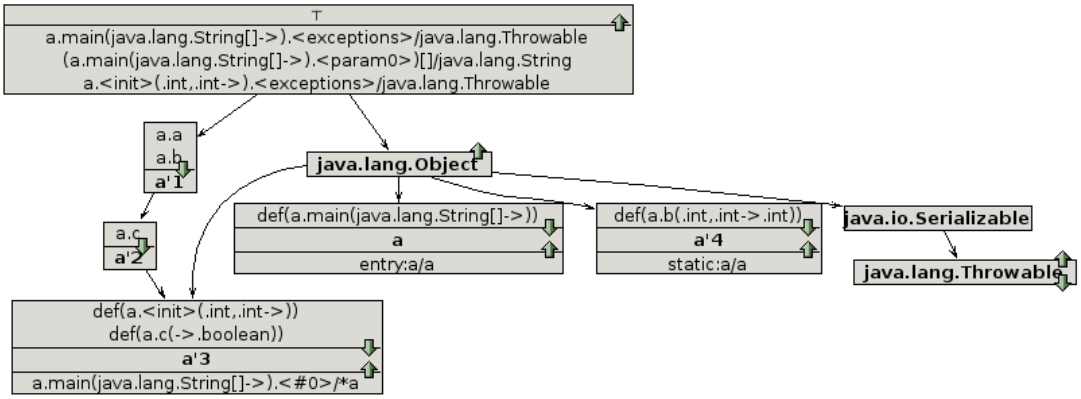

Fig. 9. Concept lattice for obfuscated code in Fig. 7

analyse small problem instances like that of Fig. 6 and 7. However, we have not tested whether KABA will be able to scale for large programs and hence we cannot conclude that Tyma's obfuscation technique cannot deliver satisfactory levels of obfuscation in some instances.

IV. Conclusion

Since Barak highlighted the need for provable security for obfuscation [21], we have witnessed a spate of obfuscation techniques based on the hard complexity problem of precisely determining alias locations at runtime [18], [3], [4], [6], [7], [5]. However, no method of generating hard problem instances is currently known, except for numeric problems such as integer factorization. No method is known for reducing the integer factorization problem to de-obfuscation.

In this paper, we have highlighted some plausible attack techniques that could be mounted on obfuscation transforms which rely on the intractability of aliasing problem as theoretical basis. At the start of this paper, we had posed two interesting research questions, viz. *What sort of analysis tools are useful for the automated understanding of the code obfuscated with aliasing transforms?* and *Can general purpose program analysis tools be used to assess the obfuscatory strength of aliasing transforms or do we need to develop customised analysis tools instead?* Throughout the paper, we tried to address these two questions with a selection of three static analysis tools (BDDBDDB, Indus, and KABA) and the analysis results they produced on test programs. We outlined a few simple techniques (like making references allocation site sensitive to thwart BDDBDDB and correlating predicates to make bigger slices) that exploit weaknesses in most advanced program analysis tools to strengthen the stealth of obfuscating transforms.

Another novel contribution of this paper is the suggestion that a relatively new analysis technique, called concept lattices, could be used in frontend tools for attacking popular obfuscations such as identifier renaming, overload induction, class coalescing and splitting. We have illustrated the use of concept lattices by analysing a simple instance of Tyma's identifier renaming obfuscation algorithm. In our future work we will construct more complicated instances of method overloading to see whether concept lattices can successfully deobfuscate these. We also hope to test the obfuscatory strength of Sosonkin's DOJ transforms [22].

References

[1] Collberg, C., Thomborson, C., and Low, D.: A Taxonomy of Obfuscating Transformations. Technical Report#148. July 1997. 36pp. Department of Computer Science, The University of Auckland, New Zealand.

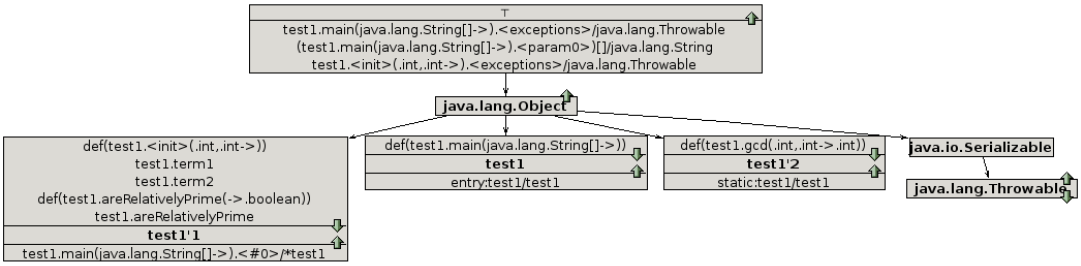

Fig. 10. Concept lattice of program in Fig. 6 after refactoring

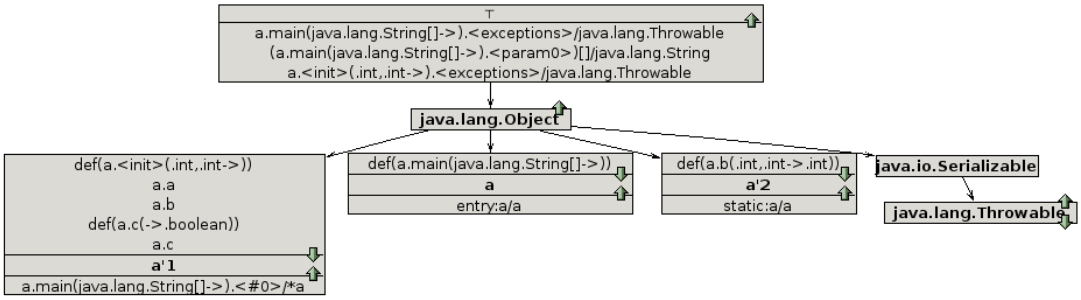

Fig. 11. Concept lattice of the obfuscated program in Fig. 7 after refactoring

[2] Collberg, C., Thomborson, C., and Low, D.: Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In Proceedings of 1998 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98). Pages 184-196. Jan 1998.

[3] Wang, C., Hill, J., Knight, J.C., and Davidson, J.W.: Protection of software-based survivability mechanisms. In Proceedings of the 2001 conference on Dependable Systems and Networks. Pages 193-202. IEEE Computer Society. 2001.

[4] Tyma, P.: Method for renaming identifiers of a computer program. US patent number 6102966. August 2000.

[5] Sosonkin, M., Naumovich, G., and Memon, N.: Obfuscation of Design Intent in Object-Oriented Applications. In Proceedings of the 2003 ACM Digital Rights Management Conference. Pages 142-153. Washington, DC, USA.

[6] Sakabe, Y., Masakazu, S., and Miyaji, A.: Java obfuscation with a theoretical basis for building secure mobile agents. In Lioy A. and Mazzocchi D. eds. Communications and Multimedia Security (CMS 2003). Volume 2828 of Lecture Notes in Computer Science. Pages 89-103. Springer-Verlag. 2003.

[7] Ogiso, T., Sakabe, Y., Soshi, M., and Miyaji, A.: Software obfuscation on a theoretical basis and its implementation. In IEICE Transactions on Fundamentals, Pages 176-186. 2003.

[8] Lhotak, O. and Hendren, L.: Scaling Java Points-To Analysis using SPARK. Sable Technical Report No. 2002-9. School of Computer Science, McGill University, Canada.

[9] Vallee-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominvolle, P., and Sundaresan, V.: Optimizing Java bytecode using the Soot framework: It it feasible?. In Proceedings of Compiler Construction (CC 2000). Pages 18-34. Volume 1781 of Lecture Notes in Computer Science. Springer-Verlag. 2000.

[10] Soot: Version 2.2.2. March 2005. http://sable.mcgill.ca/soot/

[11] Whaley, J. and Lam, M. S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In Proceedings of the ACM-SIGPLAN 2004 conference on Programming Language Design and Implementation (PLDI'04). Pages 131-144. Washington DC, USA. 2004.

[12] Nagra, J.: Thread Based Software Watermarks. PhD Thesis. Department of Computer Science, The University of Auckland, New Zealand. 2006.

[13] Tip, F.: A survey of program slicing techniques. In Journal of Programming Languages, 3(3):121-189. September 1995.

[14] Indus-0.5. March 2005. http://indus.projects.cis.ksu.edu

[15] Ganeshan Jayaraman, J. H. and Ranganath, V. P.: Kaveri: Delivering Indus Java program slicer to Eclipse. In Proceedings of Fundamental Approaches to Software Engineering (FASE'05) conference. pages 269-272. Volume 3442 of Lecture Notes in Computer Science. Springer-Verlag. Edinburgh, Scotland. 2005.

[16] Snelting, G. and Tip, F.: Understanding class hierarchies using concept analysis. In ACM Transactions on Programming Languages and Systems (TOPLAS). Pages 540-582. May 2000.

[17] Streckenbach, M. and Snelting, G.: Refactoring Class Hierarchies with KABA. In Proceedings of ACM conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'04). Pages 315-330. Vancouver, BC, Canada. 2004.

[18] Cimato, S., De Santis, A. and Ferraro Petrillo, U.: Overcoming the obfuscation of Java programs by identifier renaming. In Journal of Systems and Software, Volume 78, Issue 1, October 2005, Pages 60-72.

[19] Dotfuscator Whitepaper Version 2.0. 14pp: Accessed PreEmptive Solutions. September 2005. www.preemptive.com/documentation/dotfuscator_ whitepaper.pdf

[20] DashO Whitepaper Version 1.0. 13pp: PreEmptive Solutions. Accessed September 2005. www.preemptive.com/documentation/dasho_ whitepaper.pdf

[21] Barak, B.: Can we obfuscate programs? Accessed September 2005. http://www.cs.princeton.edu/ boaz/Papers/obf_ informal.html

[22] Naumovich.G., Yalcin, E., Memon, N., Yu, H., and Sosonkin, M.: Class coalescence for obfuscation of object-oriented software US patent number 20040103404A1. May 2004.

An extended version of this work is available with the authors and may be obtained by contacting the first author at `anirban@cs.auckland.ac.nz`