SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–28
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Effective and Efficient Java Type Obfuscation

Christophe Foket and Koen De Bosschere and Bjorn De Sutter*

SUMMARY

To protect valuable assets embedded in software against reverse-engineering attacks, software obfuscations aim at raising the apparent complexity of programs and at removing information that is useful for attackers. In this work, we propose to combine five transformations that obfuscate the type hierarchy of Java applications and eliminate much of the type information that can be inferred from the Java bytecode. We rely on some existing algorithms, present adaptations, and introduce new algorithms for some of the transformations, which are all made available in an open-source prototype implementation ready for take-up. We present an extensive experimental evaluation on benchmarks of real-world complexity, using complementary metrics that cover the protection strength against both human and tool-based reverse-engineering attack methods. The results indicate that the obfuscation is effective as well as much more efficient than the previous state of the art. For the first time, this makes these obfuscations practically viable in real-world deployment scenarios. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Java; obfuscation; overhead; type information; optimization

1. INTRODUCTION

Software often embeds security-sensitive assets with confidentiality and integrity requirements. Protecting those assets against reverse engineering and tampering by means of software protection is a complex problem. This is particularly the case for managed languages such as Java, as their bytecode format relies heavily on the presence of symbolic information and meta-information to support run-time features such as type checking, garbage collection, and reflection. Whereas that embeds assets with confidentiality requirements can be obfuscated to make it much harder to analyze and comprehend it [1], achieving high levels of obfuscation at acceptable levels of overhead remains a challenge. Improving protection techniques to lower their overhead without downgrading the resulting level of protection is therefor a relevant objective.

Consider the combined obfuscation transformations of *class hierarchy flattening* (CHF), *interface merging* (IM), *method merging* (MM), and *object factory insertion* (OFI). That combination has been shown to obfuscate the class hierarchy design and to remove a large amount of useful type information [2]. However, the published algorithms and the available prototype implementation makes applications grow up to 6 times larger, and become up to 8 times slower [2]. This is unacceptable in practice. Clearly, those levels of overhead need to be reduced.

Limiting the overhead of local obfuscation techniques such as opaque predicate insertion and function inlining to trade off the efficiency and effectiveness is straightforward: Those techniques can be deployed sparsely and based on profile information to avoid performance degradation on the most frequently executed code paths. By contrast, global design obfuscation techniques, such as the

*Correspondence to: Department of Electronics and Information Systems, Ghent University, Technologiepark-Zwijnaarde 126, 9052 Gent, Belgium. E-mail: bjorn.desutter@ugent.be

mentioned techniques that aim for hiding the whole type hierarchy of a program, typically do not offer the option of selective deployment to the user. Instead, the techniques themselves need to be adapted if we want to make them practically useful.

Such adaptations are enabled in part by language features that have been introduced after the mentioned protections were first developed, such as the default interface methods that were introduced in Java 8. Moreover, while profile information is not useful to select where to apply the transformations in the code, it offers the potential to optimize how to apply them.

In the presented research, we improved the aforementioned combination of obfuscating transformations, building on the Java 8 feature and on profile information. We experimented with variations of the core algorithms that steer the deployment of the obfuscating transformations to lower their overhead while maintaining the achieved level of protection. Furthermore, we combined them with more aggressive *identifier renaming* (IR) to make them even more efficient.

We implemented our combination of existing, adapted, and novel algorithms in a prototype tool flow that we deployed and evaluated on the DaCapo benchmark suite [3], thus covering applications of real-world complexity, using two complementary metrics to measure the effectivenss in terms of potency and resilience against human and tool-based attack methods.

Concretely, compared to the work presented before in [2], we apply IR to more identifiers, we made the introduction of so-called dummy methods during CHF optional, we use a refined decision logic for choosing candidates for MM to avoid excessive overhead, we developed a completely new OFI algorithm that exploits profile information, and our prototype tool now generates realistically sized dummy methods to prevent information leakage from the presence of empty methods.

This paper presents the algorithmic novelties, their implementation in a new proof-of-concept obfuscation tool, and experimental results on the benchmark suite. Overall, the results indicate that our improvements cut 2/3 of the worst-case size growth and 4/5 of the worst-case execution slowdowns of the pre-existing state of the art, thus bringing the amount of overhead to levels that are acceptable in many real-world scenarios. Moreover, the effectiveness of the protections is not impacted significantly, and their efficiency is made much more predictable. As such, we now consider these techniques ready for take-up in practice. To facilitate its take-up as well as future improvements by third parties, our proof-of-concept implementation is available as open source at `http://gujto.elis.ugent.be/`. It is implemented on top of Soot, a mature analysis and transformation framework for Java applications already used in many research projects [4, 5].

The remainder of this paper is organized as follows. In Sections 2 to 6, we discuss the five transformations that we combine to obfuscate the type hierarchy and type information in Java programs, clearly indicating where we reuse existing algorithms as is or where we adapted them. In Section 7, we perform an extensive experimental evaluation that includes correctness, obfuscation times, overhead, and obfuscation strength. Section 8 compares our work to related work, after which Section 9 draws conclusions and discusses interesting lines for future work.

2. IDENTIFIER RENAMING

The first transformation we deploy is identifier renaming (IR). In line with software engineering best practices, developers typically choose meaningful identifiers (i.e., names) for program elements such as classes, interfaces, methods, and fields. However, as several Java Virtual Machine (JVM) features (e.g., symbolic linking, reflection, and custom class loading) rely on symbolic information, the identifiers of all those elements need to be present in the distributed bytecode files, where they hence provide a treasure of information to reverse engineers.

An effective and efficient, widely used solution is to replace as many meaningful identifiers as possible by meaningless, randomized, and, where possible, confusing ones [6]. Confusion can be created by re-using identifiers as much as possible, rather than using different ones for each identifier. There are limits on the possible re-use, however, as re-using identifiers should, e.g., not alter how methods override one another.

IR is efficient because the length of the new identifiers can be minimized during the randomization, which results in smaller class files that will load faster. IR is also highly effective,

DOI: 10.1002/spe

```
1  class X3 extends X1{
2    X3(A a, C c, short s){
3      super(a, c);
4      X2 x2 = new X2(a, c);
5      x2.n();
6      Y2 y2 = new Y2(s, c);
7      y2.p(x2);
8      y2.p(y2);
9    }
10   void m(){
11     X1 x1 = new X1(null, b);
12     x1.m();
13     super.m();
14   }
15 }
```

(a)

```
1  class X3 implements I1{
2    X3(A a, C c, short s){
3      this(a, c);
4      I1 x2 = new X2(a, c);
5      x2.n();
6      I2 y2 = new Y2(s, c);
7      y2.p(x2);
8      y2.p(y2);
9    }
10   void m(){
11     I1 x1 = new X1(null, b);
12     x1.m();
13     m1();
14   }
15 }
```

(b)

```
1  class X3 implements I{
2    X3(A a, C c, short s){
3      this(a, c);
4      I x2 = IFactory.create(a, c, _ , _ , …);
5      x2.mrg1();
6      I y2 = IFactory.create(c, _ , _ , s, …);
7      y2.g4(x2, _);
8      y2.g2(y2);
9    }
10   void m(){
11     I x1 = IFactory.create(null, b, _ , _ , …);
12     x1.m();
13     g3(_);
14   }
15 }
```

(c)

Figure 1. (a) Original code (b) after CHF (c) after CHF, IM, MM, and OFI. We removed copied fields and methods, and dummy methods from (b) and (c). Underscores in (c) represent arguments that can be chosen arbitrarily; ellipses represent additional arguments required by the factory methods.

as it is one of the few obfuscations that permanently removes information from the distributed software, forcing an attacker to guess for names (following manual code analysis and/or unsound reconstruction by means of, e.g., recent machine learning techniques), rather than only making it harder for a reverse engineer to gather the information. IR is therefore a default obfuscation in every Java obfuscator, and for that reason alone, we include it in our combination of obfuscating transformations.

That is not the only reason to include it, however. A second major reason is that IR also allows later obfuscating transformations to become less space-inefficient. It does so by changing the ratio between space occupied by code and space occupied by symbolic information in class files, such as the identifiers occurring in the method signatures stored in the class files' constant pool entries. Some of our later transformations result in significantly longer signatures. Without renaming class and interface names, those longer signatures can become dominant in terms of space consumption. With aggressive IR, they are much less of a concern, so the later transformations can then be applied more aggressively without risking unacceptable amounts of space overhead.

This paper uses a running example to present the later transformations. Its relevant code is shown in Figure 1(a). Figure 2(a) shows the corresponding class hierarchy of two subtrees of java.lang.Object, rooted at X1 and Y1, respectively. For the sake of clarity, we do not use randomized or deliberately confusing identifiers in them. The example also refers to classes A, B, and C. The latter is a subclass of B. Those three classes are external classes, assumed to be beyond the control of the obfuscator, so their identifiers cannot be renamed, and their hierarchy cannot be transformed.

Compared to our previous work [2], we deploy IR more aggressively in this work: Whereas it was only applied to field and method names before, we now also apply it to class and interface names.

3. CLASS HIERARCHY FLATTENING

Class hierarchy flattening (CHF) removes subtype relations from an application's class hierarchy. It is a first step for obfuscating the hierarchy. A detailed description of this transformation and the preconditions to apply it conservatively can be found in previous work [2]. As we mostly reuse that existing work, here we only discuss an outline of this transformation. Figure 2 shows the flattened class hierarchy of our running example, in which classes have become siblings rather than subtypes and supertypes. Figure 1(b) shows the corresponding code.

To preserve the semantics of the application, instance fields and methods in the original program are copied from classes to their subclasses, and interface types are used to offer a common interface that would otherwise be provided through inheritance. The interface types are then used throughout

Prepared using speauth.cls
DOI: 10.1002/spe

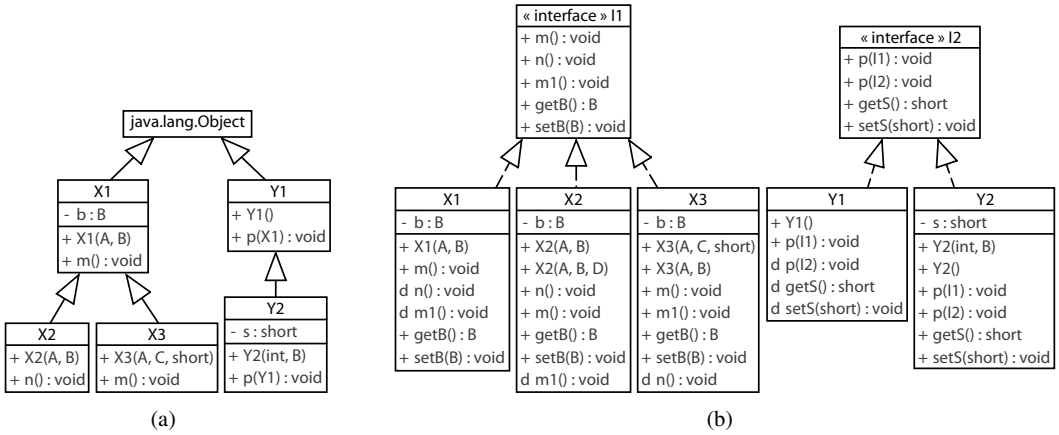

Figure 2. (a) Original class hierarchy. (b) Class hierarchy after CHF.

the application's code whenever possible; the types of variables, fields, and method parameters are replaced by their corresponding interface types to get rid of much of the original type information.

Before instance fields and methods are copied, the fields are first encapsulated to make them accessible through the interface types. In our example, methods getB/setB and getS/setS are added to X1 and Y2, respectively. During the copying, methods and fields are renamed as needed to avoid collisions with existing ones. Constructors, which have to keep their classes' names, are given an additional, distinguishing parameter. Examples include method m, which is renamed to m1 after being copied from X1 to X3, and the constructor with parameter type list [A,B], which is given an additional argument of type D after being copied from X1 to X2.

Each interface created during flattening declares all methods that are defined in the classes that implement the interface. This ensures that after the declared type of variables is changed to one of the interface types, methods can still be invoked on the objects bound to those variables. Interface I1, for example, declares method m such that m can still be invoked on x1 after the declared type of x1 is changed to I1.

Optionally, dummy methods are added to all classes or to a selected subset thereof. In Figure 2, the dummy methods are tagged with a 'd' instead of their visibility modifier. Initially, these methods are empty (except for the necessary bytecode to return a value of the proper type), and in any case their presence is not required for correctness. They can serve a purpose, however, which will become clear in the next section. The fact that dummy methods are added optionally and hence only where selected, is the only difference with the existing work [2].

The option to add or omit dummy methods from the classes exists because of the Java feature of *default interface methods*, which were added to the Java spec as of Java 8. This option allows one to define a default method in an interface, thus avoiding the need to define each interface method in each class that implements the interface.[†]

4. INTERFACE MERGING

Flattening the class hierarchy in itself does not remove all type information. From the hierarchy in Figure 2, e.g., one can still deduce that classes X1, X2, and X3 are related, since they implement the same interface. Furthermore, since each interface is implemented by a (relatively) small number of classes, much type information can still be deduced from the assignments and method signatures

[†]Interestingly, no default interface fields exist, unless they are static and final.

Prepared using speauth.cls
DOI: 10.1002/spe

used in Figure 1(b). To remove this information, we proposed interface merging (IM) in previous work [2]. We reuse it here, as is, and hence do not discuss the algorithm in detail. IM combines multiple interfaces created during flattening, resulting in fewer interfaces that are implemented by more (unrelated) classes. Figure 3 shows the class hierarchy after merging interfaces I1 and I2 from Figure 2 into a new interface I. In this version, all possible dummy methods are included.

Figure 4 shows the same hierarchy, but without any dummy class methods. It clearly shows that classes that originated from the same subtrees in the original hierarchy share more methods (in the sense of defining the same methods) than classes that originated from different subtrees. The presence of such shared methods clearly leaks a considerable amount of information about the original class hierarchy to attackers. In other words, the obfuscation so far lacks stealth.

Another potential information leak might result from identical method bodies. For example, X1::m(), X2::m(), and X3::m1() in the flattened hierarchy are all copies of the original X1::m(). To mitigate such leaks, a defender can rely on software diversification techniques to make the bodies look sufficiently different to make it impossible or at least much harder for an attacker to extract the relevant information. For native code, diversification has already proven to reduce the efficiency of automated code matching techniques (also called diffing) with orders of magnitude [7, 8]. Integrating such techniques with our transformations to thwart attacks based on code matching remains future work, as does evaluating the effectiveness of such attacks and of the defense.

Another option to mitigate attacks based on the presence and similarity of method bodies is to sow confusion by leaving some dummy class methods in place and by filling their bodies with code from unrelated classes. For example, Y1::m1() and Y1:m() can be filled with similar code as that of the original X1::m(). Doing so for all methods will result in significant code bloat, however. For that reason, it can be useful to reduce the number of methods in the class interface, as will be discussed in the next section.

Notice that also the presence of field b of external type B in certain classes hints at the fact that those classes are related. This form of information can be hidden from an attacker by inserting similar fields in the other classes. That is not without its downsides, however, as it will lead to higher dynamic memory consumption. Importantly, we need to clarify that this situation occurs relatively infrequently, because it only occurs for fields of external types. Fields of types in the hierarchy being transformed will be declared to be of the common interface type in the obfuscated program, similar to the local variable x2 in the example code fragment of Figure 1(c), which is declared to be of type I instead of X2. Such fields are then present in many of the originally unrelated classes that implement a merged interface, and they can all be named identical to maximize the confusion. Thus the correlation between classes sharing similarly typed fields and classes originating from the same class hierarchy subtrees will be weakened significantly. Again, we leave the study of attacks that use this information, and specific defenses against such attacks for future work.

5. METHOD MERGING

To minimize the number of methods declared in the merged interfaces and to reduce the number of dummy class methods that a defender might want to add to make the obfuscation more stealthy as discussed in the previous section, we propose to reuse the method merging (MM) transformation from the existing literature [2]. With this technique, two methods in an interface and in the classes that implement the interface are merged into one method by merging their signatures (i.e., merging their lists of argument types) and by merging their code bodies. As in the existing transformation, we only merge pairs of methods of which at least one is a dummy (or non-existing) method in each class, so merging the code bodies in a class comes down to keeping the non-dummy method's body (if there is one). Obviously only methods with compatible signatures are merged. For example, if both methods return a non-void type, the return type of one method has to be equal to or a subtype of the return type of the other method. The MM algorithm iteratively and greedily selects methods to merge until no more pairs of methods meet the necessary preconditions for merging. For details of the underlying algorithm and the preconditions under which the transformation can be applied, we refer the interested reader to the existing literature [2].

Prepared using speauth.cls
DOI: 10.1002/spe

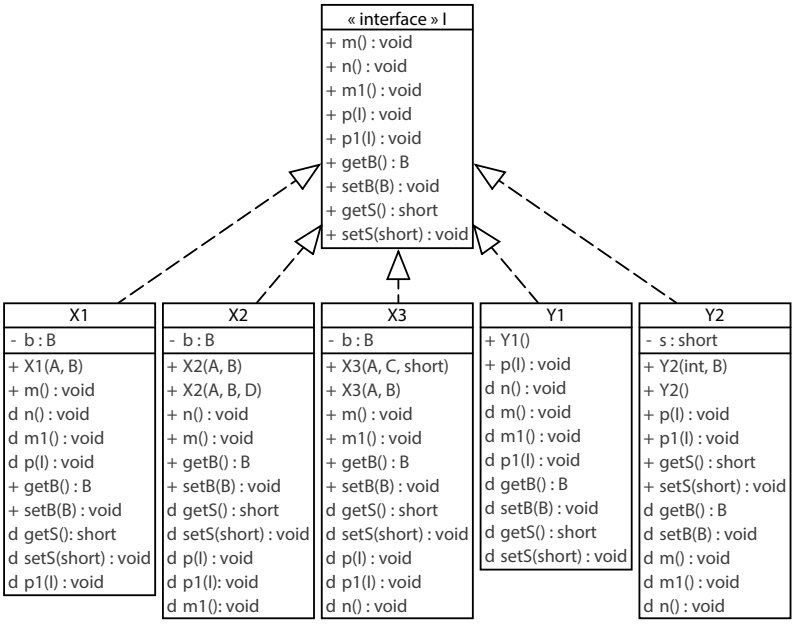

Figure 3. Class hierarchy after CHF and IM with dummy methods.

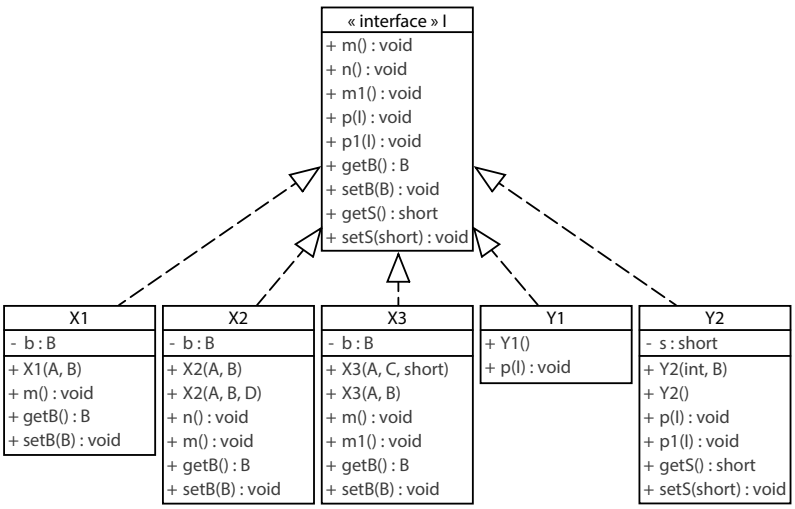

Figure 4. Class hierarchy after CHF and IM without dummy methods.

Figure 5 shows the hierarchy after MM. In this hierarchy, void n() and short getS() have been combined into short g1(), B getB() and void p1(I) into B g2(I), void m1() and void setS(short) into void g3(short), and void p(I) and void setB(B) into void g4(I,B). At this point, there are only 9 dummy methods left, which cannot be further merged into other methods. Changes to the code as a result of MM are shown on lines 5, 7, 8, and 13 of Figure 1(c). On lines 7 and 13 additional arguments are provided because g4 and g3 require more arguments than p and m1, respectively.

While our MM algorithm can merge methods under exactly the same preconditions as in the existing algorithm, we adopted a new decision logic to decide on the order of the greedy merging, on the manner in which signatures are combined, and on the stop condition. The main changes are that our new decision logic uses a cost function to stop merging methods when it starts to have a negative effect on class file size, and that it reduces the variation in signatures to avoid excessive

Prepared using speauth.cls
DOI: 10.1002/spe

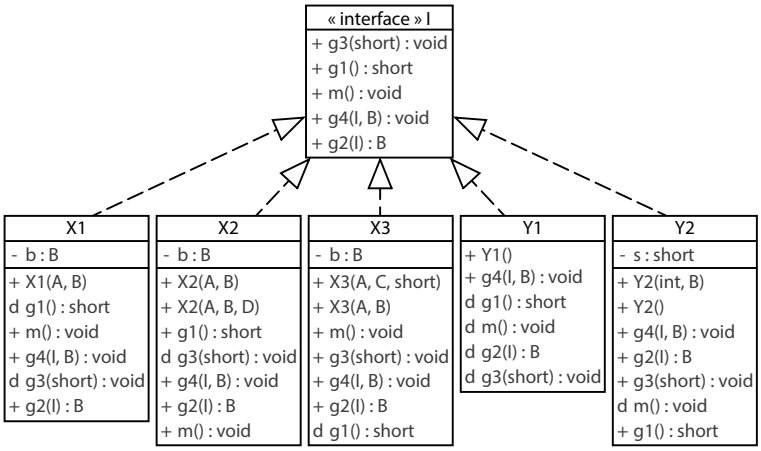

Figure 5. Class hierarchy after CHF, IM, and MM.

overhead in the form of unique constant pool entries. The details of the new algorithm are out of the scope of this paper; we refer the interested reader to Foket's PhD. thesis for a detailed discussion [9].

We wrap up this discussion by noting that all transformations discussed so far introduce relatively little performance overhead: Dummy methods are never invoked, and while MM results in longer signatures and hence in more arguments that need to be provided at method invocation sites, the overhead thereof remains relatively small.

6. OBJECT FACTORY INSERTION

After CHF, IM, and MM, type inference techniques [10] can still deduce type information from object creation sites. That type information by itself can be valuable for attackers, e.g., because it allows static analysis to compute more precise call graphs. When precise points-to sets can be computed, they can also leak information about the original type hierarchy, as objects appearing in points-to sets together more frequently are more tightly coupled in the original hierarchy.

By means of a new object factory insertion (OFI) algorithm, we aim to remove this type information originating from object creating sites. We do so by replacing constructor calls by calls to obfuscated object factories. Our new algorithm differs significantly from older ones in literature [2]. Replacing the old ones by the new one has a major impact on the overall overhead of the proposed obfuscations, so we discuss the new algorithm in detail in this section.

For maximum effect, each factory should be constructed such that it can create objects of as many different types as possible. This makes it more difficult to narrow down the exact type of the object created at different program points, i.e., where the factory methods are invoked. Conceptually, applying OFI to the running example results in the single object factory shown in Figure 6. It has a single create method that can return instances of all five classes in the application. For passing the parameters of the original constructors, enough parameters of type Object and of basic types int, long, float, and double are provided. The constructor to invoke is selected by choosing the appropriate values for a number of additional *selection parameters*, indicated by the ellipsis on line 2. For instance, to replace the object creation on line 6 of Figure 1(b), the values of these parameters should be chosen such that the expression in the if-statement on line 9 of Figure 6 evaluates to true and the others evaluate to false. To obfuscate which invocations of the factory will invoke which constructors, existing obfuscation techniques can be used, such as opaque predicates [11] and mixed boolean-arithmetic expressions [12]. The values for factory method parameters that are not passed to the selected constructor can be chosen arbitrarily, as shown with underscores on line 6 of Figure 1(c).

DOI: 10.1002/spe

```
1  class IFactory {
2   static I create(Object o1, Object o2, Object o3, int i, ...) {
3     if(...) return new X1((A)o1, (B)o2);
4     if(...) return new X2((A)o1, (B)o2);
5     if(...) return new X2((A)o1, (B)o2, (D)o3);
6     if(...) return new X3((A)o1, (C)o2, (short)i);
7     if(...) return new X3((A)o1, (B)o2);
8     if(...) return new Y1();
9     if(...) return new Y2(i, (B)o1);
10    return new Y2();
11  }
12 }
```

Figure 6. Object factory with a single factory method.

```
1  class IFactoryImproved {
2    static I create(Object o1, Object o2, Object o3, ...) {
3      if(...) return new X1((A)o1, (B)o2);
4      if(...) return new X2((A)o1, (B)o2, (D)o3);
5      if(...) return new X3((A)o1, (B)o2);
6      if(...) return new Y1();
7      return new Y2();
8    }
9    static I create(Object o1, Object o2, int i, ...) {
10     if(...) return new X1();
11     if(...) return new X2((A)o1, (B)o2);
12     if(...) return new X3((A)o1, (C)o2, (short)i);
13     if(...) return new Y1();
14     return new Y2(i, (B)o1);
15   }
16 }
```

Figure 7. Object factory with two factory methods.

Note that the casts on lines 3–7 and on line 9 of Figure 6 are needed to cast the arguments of the factory method, which are passed as instances of type java.lang.Object and values of type int, to the types A, B, C, D, and short that are expected by the original constructors. These casts reveal some type information. However, the information from static analsysis will only be precise locally, within the factory method. If the aforementioned obfuscation of the selection parameters is not undone, all inferred type information will be merged by the analysis at the boundaries of the factory method, and hence no precise type information will leak outside that method.

The factory shown in Figure 6 is just one of the possible factories the new algorithm can generate. If, based on profile information that is collected ahead of time during the execution of the untransformed application, the algorithm computes that the factory from Figure 6 would result in too much execution time overhead, it may instead generate the less expensive factory shown in Figure 7. In this factory, calls to the original constructors are distributed over two different factory methods. Also, an additional call to a newly created no-argument constructor for class X1 is added on line 10 to maximize the number of different types of objects that can be created by the factory.

To construct factories such as the ones in Figures 6 and 7, our algorithm operates in three steps. In the first step, information about object creation sites in the program is gathered to determine the properties of the object factories. While collecting this information, each constructor is modeled as a separate low-overhead factory method. In the next step, these simple factory methods are merged into increasingly more complex ones until a user-defined overhead threshold is reached. In the last step, a new class is created for each object factory, and constructor calls are rewritten to invocations of the factories' methods. Each step is explained in more detail below. The first two steps are fundamentally different from the algorithm in our previous work [2], as the old algorithm targeted only a single, huge factory method per factory class. The old algorithm hence did not involve iterative merging of smaller factory methods as in step two of the new algorithm and hence also did not require the collection of useful information to steer the merging process as is done in step one of the new algorithm. The creation of actual factory classes and methods and the rewritting of object creations in step 3 is similar to our previous work, albeit that more factory methods are created in the new algorithm.

DOI: 10.1002/spe

It is worth noting that the potency of an object factory creation algorithm greatly depends on its ability to create generic object factories that can instantiate many different types of objects. How generic these can be depends on the types assigned to local variables in a program's intermediate representation [2]. Indeed, when more abstract types are assigned to local variables, this relaxes the constraints on which types of objects a factory can return such that the returned objects can be stored in those variables. As a result, the factories can be more generic, which means they have a greater potential at confusing pointer analyses, including points-to analysis, alias analysis, and call graph reconstruction. In what follows, we therefore assume that the program has been pre-processed using a type inference that assigns to each local variable in the program's intermediate representation the most abstract type possible. For this, we reuse the type inference algorithm from our previous work [2], which is a slightly adapted version of the algorithm by Gagnot et al. [13].

6.1. Collecting information

Our algorithm starts by building a key-value pair mapping $\mathfrak{F}$ that contains information about all object creations and constructors in the program, as well as the object factory classes and methods to be generated. Each key in the mapping consists of a 2-tuple (r, D) representing a factory class, where r is the return type of the factory's methods, and D is the set of all classes whose constructors the factory should be able to call. Each tuple (r, D) in $\mathfrak{F}$ maps to a list F of tuples that each contain information about one of the factory's methods. Each tuple $(K, O) \in F$ consists of a list K of constructors the factory method should call, and a set O of object creations that should be replaced by calls to the factory method. Given an application, the mapping $\mathfrak{F}$ is constructed as follows:

- Let $\mathbb{A}$ be the set of all application classes and interfaces, $\mathbb{L}$ the set of all library classes and interfaces, and $\mathbb{T}$ the set of all types.
- Let $a : (\mathbb{A} \cup \mathbb{L}) \mapsto \mathcal{P}(\mathbb{A} \cup \mathbb{L})$, be a function where $a(t)$ gives the set of all classes and interfaces assignment-compatible with t.
- Let $j : (\mathbb{A} \cup \mathbb{L}) \mapsto \mathcal{P}(\mathbb{A} \cup \mathbb{L})$, be a function where $j(t)$ gives the set of all classes and interfaces located in the same directory or archive as t.

For each object creation o : `x = new C(...)` in the application calling a constructor k, where the declared type of `x` is `X` and `C` $\in \mathbb{A}$, our algorithm determines the properties of the corresponding object factory. To do this, it constructs the set of potential factory return types as

$$R = \{r \in \mathbb{A} \cap a(\mathsf{X}) \cap j(\mathsf{C}) \mid \mathsf{C} \in a(r)\}.$$

It then computes the return type r as the most abstract in R, i.e., the type with the most subtypes:

$$r = \arg\max_{r_i \in R} |a(r_i) \cap j(\mathsf{C})|.$$

It may seem strange to compute r in this manner, or even that r has to be computed at all, as `X` clearly is the best choice for the return type. After all, this type is the most abstract type that could be assigned to `x`. It hence has more subtypes than any other type in $a(\mathsf{X})$, and is therefore the best choice to create factory methods that can return many different types of objects.

However, in practice it is not always possible to choose `X` as the return type. The reason is that doing so blindly might lead to class loading errors. By default, we can only assume that when the object creation o is executed, class `C` has been loaded. In other words, that there is a class loader that, before executing o, can load class `C`, and other classes located in the same directory or archive as `C`[‡]. Hence, when o is replaced by a call to a factory method, we must ensure that the return type of the factory method, the class declaring the factory method, and the declaring classes of constructors called by the factory method are located in the same directory or archive as `C`, such that they can be

[‡]Note that it is possible to write custom class loaders for which this does not hold. While we did encounter custom class loaders during our experiments, we did not encounter ones that exhibited such behavior.

DOI: 10.1002/spe

loaded by the same class loader that loads C. The return type is therefore chosen as the type that is assignment-compatible with X, that is located in the same directory or archive as C, and that has the most subtypes in that directory or archive. The latter condition maximizes the set of classes whose constructors the factory should be able to call. Once r has been determined, this set of classes can be computed as $D = a(r) \cap j(\mathsf{C})$.

The tuple (r, D) now uniquely identifies the factory class that will contain the method that should be invoked to replace the object creation o. This tuple essentially contains the same information as is stored in the variable R and computed by the expression $q(R)$ that is used in the old algorithm for OFI [2]. However, whereas the old algorithm used each such tuple to create a single factory method that invoked all constructors, the new algorithm uses this information to generate multiple factory methods that each invoke a single constructor. These methods are then later merged into fewer factories that each invoke multiple constructors. To add the information in (r, D) to $\mathfrak{F}$, the algorithm operates as follows:

1. Let F be the list of tuples for key (r, D) in $\mathfrak{F}$. If $\mathfrak{F}$ does not contain a mapping for (r, D), create it as follows:
   (a) Create a new empty list F.
   (b) For each element $k_i \in K_D$, the set of all constructors in all classes in D
      i. Create a new singleton $K = \{k_i\}$.
      ii. Create a new empty set O.
      iii. Add the tuple (K, O) to F.
   (c) Map (r, D) to F in $\mathfrak{F}$.
2. Find the tuple (K, O) in F for which $K = \{k\}$, and add o to O.

Note that since the actual factory methods have not yet been created, step 2 just adds information to $\mathfrak{F}$ indicating that o needs to be replaced by a call to whichever factory method invokes k.

After this part of the algorithm has finished, each tuple (K, O) represents a factory method that invokes a single constructor. In the next section, we discuss how to merge pairs of these tuples to create more effective factory methods that invoke multiple constructors.

6.2. Merging factory methods

Algorithm 1 shows the greedy algorithm we developed to merge factory methods. It makes use of the following definitions.

- Let e be the function for which $e(o)$ gives the number of times the object creation o is executed during an average run of the untransformed application. The data returned by this function is computed once ahead of time.
- Let $u : \mathbb{T} \mapsto \mathbb{T}$ be the function that maps all reference types to java.lang.Object, the primitive types boolean, byte, char, short, and int to int, and every other type to itself.
- Let $\mathbb{K}$ be the set of all constructors in the application.
- Let $p : \mathbb{K} \mapsto \mathbb{T}^*$ be the function for which $p(k)$ gives the parameter type list of k[§], and let $p_u : \mathbb{K} \mapsto \mathbb{T}^*$ be the function for which $p_u(k)$ gives the parameter type list $p(k)$ in which each type t has been replaced with $u(t)$.
- Let $m_u : \mathcal{P}(\mathbb{K}) \mapsto \mathbb{T}^*$ be the function for which $m_u(K)$ gives the parameter type list obtained after merging all parameter type lists in the set $\{p_u(k) \mid k \in K\}$.
- Let a be the number of arguments each factory method uses to determine which constructor to invoke.

[§]For a given set $\mathbb{T}$, we use $\mathbb{T}^*$ to denote variable-length tuples of elements of $\mathbb{T}$, similar to how $\mathbb{T}^n$ is used in mathematics to denote n-tuples of elements of $\mathbb{T}$.

DOI: 10.1002/spe

ALGORITHM 1: Factory method merging.

$a_{\text{con}} = a_{\text{fac}} = 0$
foreach $(r, D) \mapsto L \in \mathfrak{F}$ **do**
  foreach $(\{k\}, O) \in L$ **do**
    foreach $o \in O$ **do**
      $a_{\text{con}} = a_{\text{con}} + (|p(k)| + 1) \cdot e(o)$
      $a_{\text{fac}} = a_{\text{fac}} + (|p(k)| + a) \cdot e(o)$
    end
  end
end
$a_{\text{fac}} = a_{\text{fac}} + a_{\text{con}}$

$\mathfrak{K} = \{(r, D, l_1, l_2) \mid l_1, l_2 \in \mathfrak{F}(r, D) \wedge l_1 \neq l_2\}$
while $\exists\, \mathfrak{k} \in \mathfrak{K} \,.\, a_{\text{fac}} + c(\mathfrak{k}) \leq a_{\text{con}} \times \tau$ **do**
  $\mathfrak{k} = \arg\min_{\mathfrak{k}_i \in \mathfrak{K}} c(\mathfrak{k}_i)$
  $a_{\text{fac}} = a_{\text{fac}} + c(\mathfrak{k})$
  merge_update$(\mathfrak{K}, \mathfrak{k})$
end

To limit the overhead of the factory methods, the algorithm is controlled by a user-defined threshold τ that represents the maximum ratio by which the dynamic number of arguments required to create objects is allowed to increase. To determine when this threshold is reached, the algorithm starts by computing the dynamic number of arguments required when creating objects by invoking the constructors directly, and when creating objects using the factory methods created in the previous step. These numbers are stored in the variables a_{con} and a_{fac}, respectively. When computing a_{con}, the algorithm computes the number of arguments of a constructor k as $|p(k)| + 1$ rather than $|p(k)|$ to account for the object being initialized by the constructor also being passed as an implicit argument. Note that a_{fac} is computed in two steps. First, the algorithm computes a_{fac} as the dynamic number of arguments that must be passed to the factory methods, after which a_{con} is added to a_{fac} to count also the arguments that are passed from the factory methods to the constructors.

After computing a_{con} and a_{fac}, the algorithm constructs the set $\mathfrak{K}$ of tuples that correspond to possible factory method merge operations. Each tuple (r, D, l_1, l_2) consist of a pair (r, D) that identifies a factory class, and two elements l_1 and l_2 that each contain information about one of the factory's methods that will be merged. The algorithm then continues by greedily and iteratively selecting tuples from $\mathfrak{K}$ that will result in the smallest increase in the dynamic number of arguments required to create objects. This increase is computed by means of the cost function c, which is defined as follows:

$$\begin{aligned} c(\mathfrak{k}) = c(r, D, l_1, l_2) &= c(r, D, (K_1, O_1), (K_2, O_2)) \\ &= (|m_u(K_1 \cup K_2)| - |m_u(K_1)|) \sum_{o_1 \in O_1} e(o_1) \\ &+ (|m_u(K_1 \cup K_2)| - |m_u(K_2)|) \sum_{o_2 \in O_2} e(o_2). \end{aligned}$$

After a merge operation has been selected and executed, the set of possible merge operations is updated with the merge_update subroutine. This routine performs the following steps when provided with a reference to the set $\mathfrak{K}$ and a tuple $\mathfrak{k} = (r, D, l_1, l_2)$.

1. Create a new tuple $l_m = (K_1 \cup K_2, O_1 \cup O_2)$, where $(K_1, O_1) = l_1$ and $(K_2, O_2) = l_2$.
2. Remove l_1 and l_2 from $\mathfrak{F}(r, D)$.
3. Remove from $\mathfrak{K}$ all tuples involving l_1 or l_2.
4. Add a new tuple (r, D, l, l_m) to $\mathfrak{K}$ for each $l \in \mathfrak{F}(r, D)$.
5. Add l_m to $\mathfrak{F}(r, D)$.

DOI: 10.1002/spe

6.3. Generating factory classes

In this final step, the algorithm generates the factory classes based on the information in $\mathfrak{F}$, and replaces constructor calls by calls to those classes' methods. For each entry $(r, D) \mapsto L$ in $\mathfrak{F}$, it proceeds as follows:

1. Add a no-argument constructor to each class in D that does not already has one.
2. Create a new class f with a unique name in the directory or archive that contains type r.
3. For each tuple $(K, O) \in L$
   (a) Compute D' as the set of classes that declare the constructors in K, and add the no-argument constructor of each class $d \in D \setminus D'$ to K.
   (b) Initialize a new parameter type list p to $m_u(K)$.
   (c) Extend p with types corresponding to the parameters that will be used to decide which of the constructors in K to invoke.
   (d) Create a new factory method m in f with return type r and parameter type list p. The body of f contains calls to all constructors in K, as well as logic to decide which constructor to invoke based on the values of the parameters added in (c). The arguments to the constructors are cast from their mapped types $u(t)$ to their required types t.
   (e) Replace all object creations in O with calls to m. Provide necessary dummy arguments.

For the running example from Figure 1(a) the new algorithm may generate the factory class shown in Figure 6, as described above. The create method of this class requires three parameters of type java.lang.Object because of the constructor of class X2. If that constructor required fewer arguments, the factory method would also require one argument less. To keep the number of factory method parameters low, it is therefore important that each constructor only requires as many parameters as absolutely necessary. In light of this, the next section discusses how CHF as presented in Section 3 can be improved to reduce the number of cases in which a distinguishing parameter has to be added to the constructors during flattening.

6.4. Improving class hierarchy flattening

CHF in combination with OFI may sometimes result in factory methods that require more arguments than strictly necessary. This is because during subtree flattening, potential constructor collisions are avoided by adding artificial, distinguishing parameters to the constructors being copied from the superclasses to the subclasses. As a new, better alternative, we propose to resolve collisions by first trying all possible permutations of a constructor's parameter type list before adding a new distinguishing parameter to it. In general, for a constructor with parameter type list p, the number of permutations that can be tried before a new parameter needs to be added is given by

$$\frac{|p|!}{\prod_{i=1}^{n} f_i!},$$

where $\{t_i \mid \forall i \in [1, n]\}$ is the set of types occurring in p, and f_i is the number of times each type t_i occurs in p. In many cases, the number of permutations is large enough to avoid having to add an additional parameter.

As an example, consider the constructor X1(A,B) from Figure 2(a). When this constructor is copied to class X2 using the original class hierarchy flattening algorithm, an extra parameter of type D is added to it, because class X2 already declares a constructor with parameter type list [A,B]. Our improved algorithm for CHF, by contrast, will not add a distinguishing parameter if there exists a permutation of the parameter type list [A,B] for which X2 does not have a corresponding constructor. This is the case for permutation [B,A], so constructor X1(A,B) will be copied as X2(B,A) instead of X2(A,B,D). As a result, OFI will be able to create a factory method that requires one parameter of type java.lang.Object less than the factory method shown in Figure 6.

DOI: 10.1002/spe

Note that the technique presented here can be extended so that it can serve as a generic preprocessing step for OFI that removes unnecessary arguments from constructors, reordering their arguments to avoid collisions if necessary. However, as the usefulness of this extension may be limited, we decided not to implement it.

7. EVALUATION

7.1. Proof-of-concept implementation - experimental setup

Our proof-of-concept implementation that combines the five discussed transformations of IR, CHF, IM, MM, and OFI in one tool builds on Soot [4, 5], a mature and popular program analysis and transformation framework for Java applications. The tool rewrites bytecode packaged in a collection of jar files; it does not require access or changes to source code.

Clearly, our transformations build on a closed-world assumption, as the whole program needs to be available for computing points-to sets. To detect the set of non-transformable classes and to ensure that Java features such as reflection and custom class loading are handled correctly in our experiments, we relied on the TamiFlex Play-out Agent, a tool developed specifically for enabling static analysis of Java programs that use such features [14]. This profile-based tool relies on the developer to provide inputs that generate sufficient coverage of the whole program.

We know of no automated analysis that enables safe transformations in the presence of arbitrary class loaders. So we impose three restrictions on applications eligible for our type obfuscations. First, the class loader hierarchy of the applications does not change dynamically. Secondly, each class loader only loads classes and interfaces of which the definition is known at obfuscation time. Finally, each class loader only loads classes from a fixed set of directories, jars or other archives. With these restrictions, we can determine exactly in which directory or jar to insert new classes and interfaces such that all of them will be loaded by the correct class loader.

By inserting interfaces and flattened classes into the existing jars, rather than in new jars, the obfuscated application does not need to be combined with class loading intervention tools such as the TamiFlex Play-in Agent. Our obfuscated applications are hence as self-contained as the original ones. This eases the take-up of our results in industry, as it makes it easier to integrate our tool in existing software development life cycles (SDLCs).

Our tool can generate dummy methods with arbitrarily-sized bodies. As it is only a proof-of-concept implementation, and as we leave the study of matcher-based attacks for future work, the dummy methods are currently simply filled with `nop` instructions. Even though this approach will generate dummy methods that look far from realistic, it is sufficient to determine what the overhead in terms of application size would be if the dummy methods were filled realistically, without having to resort to more complex algorithms that actually generate realistically-looking code.

One might argue that this is not the case, as realistic methods will likely contain instructions that refer to classes, methods and fields. Hence, one could expect that filling a method body with realistic code would increase the size of the class containing that method by more than just the instructions that make up the body. Indeed, references to various other classes, methods and fields also need to be stored as constant pool entries within the class. However, in our case we do not really need to worry about this, as classes containing dummy methods typically already contain many constant pool entries that could simply be reused by the instructions injected into the dummy methods in a realistic scenario. As a result, we do not expect that filling the dummy methods in a more realistic way will result in significant additional increases in application size.

Whenever we include dummy methods in an obfuscated benchmark version, we make all dummy method bodies 66 bytes large. This is the average method size as observed by Collberg [15]. The total space occupied by our filled dummy methods is therefore equal to the space that the most stealthy bodies would occupy.

For evaluating the effectiveness of the proposed obfuscations, we relied on the robust and configurable T.J. Watson Libraries for Analysis (WALA, http://wala.sf.net) to compute points-to sets [16]. To ensure that WALA's call graph construction includes the program parts that are

Prepared using speauth.cls
DOI: 10.1002/spe

reachable through reflection, we ran WALA on benchmark versions in which TamiFlex Booster had replaced indirections through reflection by direct invocations [14].

For evaluating performance overheads, we measured execution times on computing nodes consisting of dual socket Intel Xeon E5-2670 processors with 64 GB of memory, using the Java SE Runtime Environment (build 1.8.0_162-b12) and the Java HotSpot 64-bit Server VM (build 25.162-b12).

7.2. Correctness

Regarding the TamiFlex Play-out Agent and Booster, we point to the literature for a discussion of their validity [14].

To assess the correct implementation of all transformations, i.e., to increase confidence in their soundness and preservation of program semantics, we checked that all benchmark versions passed type verification and produced correct output. These checks included many more benchmark versions than the ones on which we report results later in this paper.

In our Soot-based tool, our obfuscations are to a large extent implemented as combinations of refactorings, which are quite similar to existing refactorings [17, 18, 19]. Tip et al. express the valid refactoring space by means of type constraints [18]. Based on the original program's code and points-to sets, a set of type constraints is constructed that determines the freedom to alter declarations in the program without affecting type correctness and without changing the program's functionality, taking into account the interfaces with external libraries that cannot be rewritten, occurrences of overriding, the dynamic behavior of casts and array stores, etc. From this original set of constraints, a new set of constraints is derived that needs to be met by a refactored program. For advanced refactorings that involve code duplication and/or replacing classes by other classes with equivalent functionality, the new set of constraints allows original methods and classes to be replaced by their new counterparts while still meeting all constraints related to type-correctness, libraries, and all dynamic program behavior. While we did not implement a type constraint system and solver as done by Tip et al., we did carefully check that the limitations imposed on our obfuscations, e.g., with respect to external library types, are in line with the constraints imposed by Tip et al.

We also checked that the bytecode produced by our tool, including the rewritten cast operations and merged methods, meets all requirements for maintaining program behavior, i.e., that at all places and at all times, the same exceptions will be thrown as in the original program, and that the same or equivalent (e.g., merged) methods are invoked.

Finally, with the class loader restrictions mentioned above we can ensure that each flattened class is loaded by the exact same class loader that originally loaded its unflattened counterpart. As our obfuscations don't require any changes to where code is loaded from, who signs code (if anyone), and what default permissions are granted, this ensures that all security policies, domains, and permissions implemented for the original application by means of the Java SE Platform Security Architecture [20] remain intact. This further eases the integration of our techniques and tools into existing SDLCs.

7.3. Benchmarks

We evaluated our prototype implementation on the benchmarks shown in Table I. All benchmarks are taken from the *9.12-bach* release of the *DaCapo* benchmark suite [3]. The benchmarks in this suite closely resemble real-world applications, not only in size, but also in structural complexity, and in the set of Java language features they make use of. One notable aspect of the benchmarks is their varying amount of transformable classes. Classes that are the subject of reflection or custom class loading cannot be transformed by our tool. The case of jython especially jumps out. This is a Python interpreter that dynamically generates Java classes for the Python code it interprets. As we cannot adapt that highly input-dependent dynamic code generation, we cannot transform the static jython classes referenced by the dynamically generated classes either.

Below we present results obtained by applying different obfuscator configurations on the benchmarks. As a baseline for all comparisons, we use the original version of the benchmark from

DOI: 10.1002/spe

Table I. Overview of DaCapo 9.12-bach benchmarks.

	# application		# transformable	application size (MB)		obfuscation time (min)	
	classes	interfaces	classes (CHF)	pre IO	post IO	class meth.	default meth.
avrora	1836	83	1657 (90%)	4.1	2.3 *(-44%)*	17.2	9.8
batik	3787	856	3383 (89%)	12.5	8.8 *(-30%)*	54.9	33.8
eclipse	5213	1261	3886 (75%)	25.7	18.5 *(-28%)*	135.7	55.2
fop	4033	446	3105 (77%)	11.0	7.9 *(-28%)*	65.1	43.0
h2	1843	78	1454 (79%)	9.3	7.1 *(-24%)*	32.7	18.8
jython	3702	166	941 (25%)	11.8	10.8 *(-8%)*	24.2	10.9
luindex	605	28	510 (84%)	1.9	1.2 *(-37%)*	6.0	3.2
lusearch	608	28	510 (84%)	1.9	1.2 *(-37%)*	6.4	3.1
pmd	1999	451	1507 (75%)	5.6	4.4 *(-21%)*	26.6	15.4
sunflow	679	59	557 (82%)	2.0	1.4 *(-30%)*	10.8	7.3
tomcat	2173	268	1538 (71%)	10.1	7.0 *(-31%)*	69.2	37.2
xalan	2460	426	2111 (86%)	9.6	7.3 *(-24%)*	38.1	21.0

the DaCapo suite with class names, interface names, field names, and method names obfuscated, and shortened as a result. This enables a meaningful, realistic evaluation with respect to application size, as any real-world use of Java obfuscation also includes identifier obfuscation. For the sake of completeness, the fifth and sixth column of Table I show the space savings on the original benchmarks obtained because of IR. It can be observed that the class file reductions vary between approximately 21% and 44%, with the exception of jython. Its size reduction is much smaller because the identifiers referenced by dynamically generated code cannot be renamed.

For a second configuration, we generated benchmark versions using CHF and OFI, but without IM, and hence also without MM.

For five more configurations, we used CHF, MM, OFI, and IM with thresholds 10, 20, 30, 40, and 50, respectively. Implicitly, the use of IM always includes MM as well. The threshold for IM indicates how aggressively it should be applied. It corresponds to the maximum number of classes by which each interface can be implemented before merging stops. Instinctively, higher thresholds imply stronger obfuscation. For all configurations involving OFI, we set τ to 3.

For each benchmark we thus always report six relative numbers, for the six described configurations relative to the baseline version. For each configuration except the baseline, we generated ten different versions of each benchmark using ten different random seeds. The random seeds are used to group interfaces during IM, and to break ties during MM and OFI. Below, we always report average results for those ten versions.

To asses the scale of the trade-off that has to be made between code size and stealth by using less or more default methods for dummy methods, we performed experiments in which all dummy methods are implemented by means of default methods, and experiments in which none are implemented as default methods, i.e., all are implemented as class methods.

7.4. Obfuscation times

As shown in the right-most column of Table I, it takes our obfuscator tens of minutes to generate a single obfuscated version at maximum obfuscation settings for most benchmarks. When dummy methods are implemented as class methods, it takes even more than two hours to obfuscate eclipse. Using default interface methods reduces the obfuscation time by 44% on average. This is due to the smaller call graphs: When dummy methods are implemented as class methods, those methods do end up in the call graphs that the obfuscator builds as part of the points-to analysis on which CHF relies. This happens despite the dummy methods being unreachable, because the points-to analysis is not precise enough to determine their unreachability. Default methods also end up in the call graphs, but there are of course much fewer of them.

These high obfuscation times are obviously an issue. In the future, we plan to research how to speed up the obfuscation without reducing their effectiveness. CHF currently accounts for 70-80% of the total obfuscation time. This is mostly because of the (slow) pointer analysis. As CHF requires up-to-date pointer information at different steps during the transformation, one way to reduce obfuscation time could be to update the pointer information during flattening rather than to

Prepared using speauth.cls
DOI: 10.1002/spe

recompute it multiple times as is done in our current obfuscator implementation. However, these and other optimizations to reduce the transformation time of CHF and IM are outside the scope of this paper.

7.5. Application size and execution time overhead

Figures 8 and 9 show the sizes of the obfuscated applications relative to the size of the baselines. The former shows the minimal obtainable size when all dummy methods are implemented as default interface methods. The latter puts an upper bound on the sizes that would be obtained when all dummy methods are implemented as stealthy class methods. It is clear that in the most extreme case, those methods lead to an unacceptable class file size explosion, in particular when interfaces are merged more aggressively.

How many dummy methods should be added as class methods to make the obfuscation sufficiently stealthy remains future work, as previously indicated. We conjecture that relatively few of them will already do quite a good job¶, but more research is needed to confirm this. If our conjecture is correct, the expected code size overhead of our proposed combination of obfuscations will be the one shown in Figure 8. This will of course also be the case for defenders that simply do not care about custom attacks based on the correlation of methods present in flattened classes, e.g., because such attacks can as of yet not be executed by script kidies or amateurs using existing tools out of the box. The overhead is still significant, but in many practical cases we believe it can be acceptable.

Interestingly, the size overheads depicted in Figure 8 decrease when more IM is deployed. The reason is that parameters of merged methods are declared as the most abstract type possible, which most often is a merged interface. With increasing amounts of IM, the number of interfaces decreases, and with it the number of different types that need to be available in the signatures of merged methods. The merged signatures can hence be implemented with less parameters, and less dummy objects need to be passed at their call sites.

Figure 10 shows the median relative execution time of obfuscated versions of our benchmarks (with default interface methods) during steady-state execution. Similar to Kalibera et al. in some experiments in their paper on rigorous benchmarking [21], we consider 9 warm-up iterations, and measure the 10th iteration of each benchmark within the same VM execution. In this iteration, the bytecode is no longer interpreted, and the JVM has performed most if not all of its just-in-time compiler optimizations. As a result, the execution times measured in this phase exclude VM and application initialization overheads. Notice that execution times are nearly identical when class dummy methods are used instead of default interface methods, because the dummy methods are never invoked and hence never even get loaded into the interpreter or compiler.

For most benchmarks, the execution time overhead is well below 20%, which can be acceptable in many real-world scenarios. For eclipse, fop and h2, the execution time overheads are considerably higher, however, at around 30%, 55% and 65% respectively. This will likely still be considered unacceptable in many deployment scenarios.

The size and execution time overheads reported in Figures 8 and Figure 10 are much lower than the overheads previously reported in literature for comparable obfuscations with the exact same goal of hiding type information and the type hierarchy [2]. For eclipse the previously reported performance overhead was up to a factor 10 in the worst case, for fop it was up to a factor 3.2, and for h2 it was up to a factor 4.7. So for one benchmark, eclipse, the contributions presented in this paper were able to reduce the execution time overhead with 97%. The contributions decreased the overall worst-case execution time overhead with approx. 80%.

¶This wording is rather fuzzy on purpose: Software protection against so-called man-at-the-end attacks (that comprise reverse engineering and tampering) is by definition fuzzy. Its goal being is to delay an attack as much as possible, with the best-case outcome that the protection makes the attacks so hard that attackers give up [1]. The goal is not to prevent attacks completely, as that is in general impossible given the whitebox access that MATE attackers have to the protected software in their labs. With respect to stealth and its role in hampering MATE attacks, no formal metrics exist, and no well-defined statements can be made as a result.

DOI: 10.1002/spe

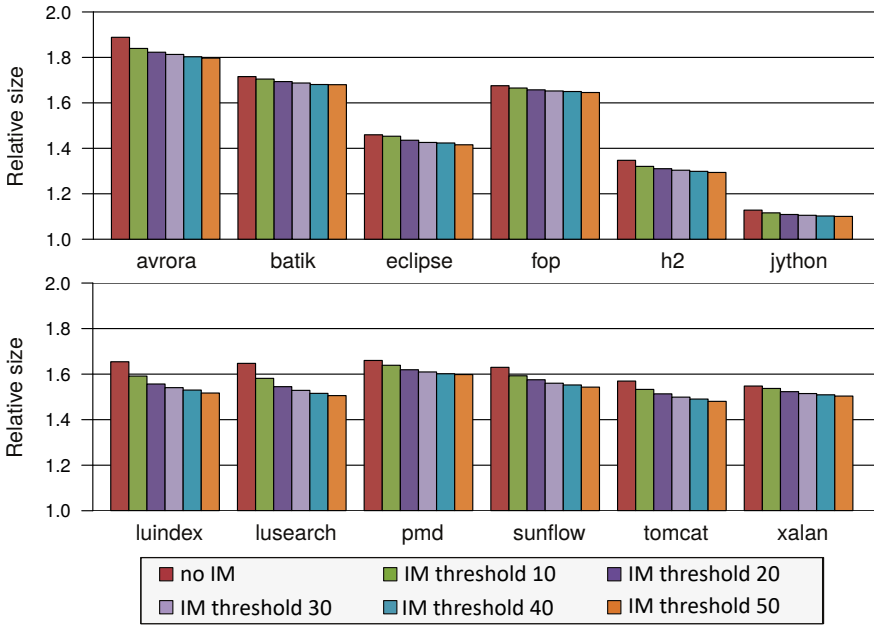

Figure 8. Relative application size of obfuscated benchmarks, with maximal use of default interface methods.

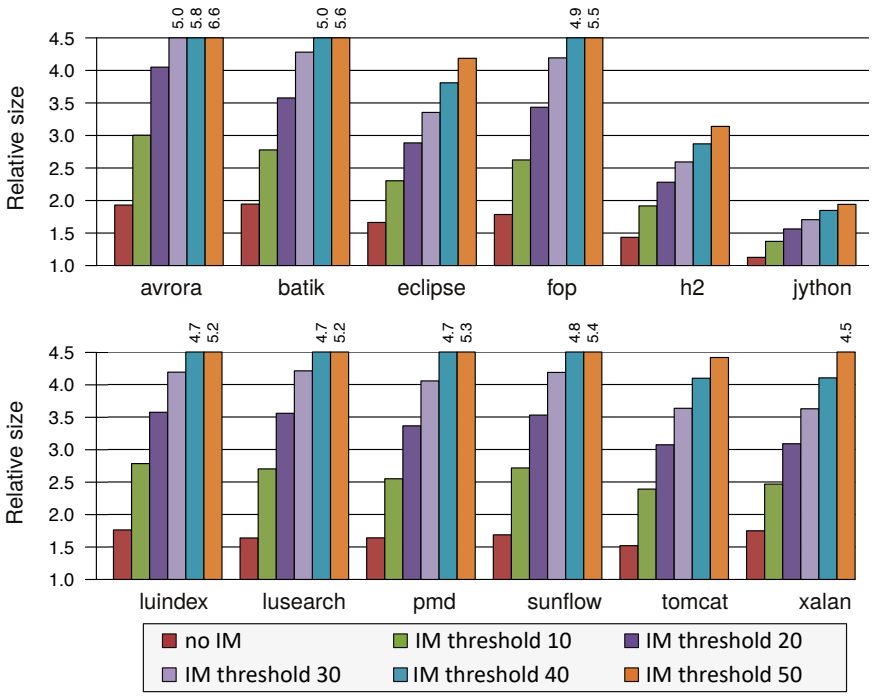

Figure 9. Relative application size of obfuscated benchmarks, without use of default interface methods, i.e., with maximal use of class dummy methods.

Most of this performance improvement results from the new algorithm for profile-based OFI, as factory method parameter type lists are on average 65% shorter than in the existing work. This reduction in signature sizes is also to some extent responsible for the much lower application size overhead. The previously reported size overheads went all the way up to a factor 6.2 for eclipse. For most benchmarks, application size grew with around a factor 5 with maximal IM. With the

DOI: 10.1002/spe

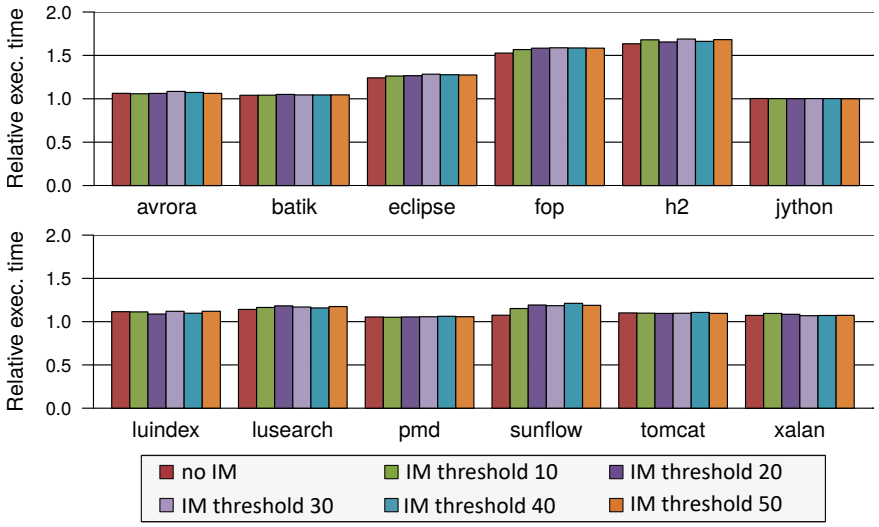

Figure 10. Relative execution time of obfuscated benchmarks with maximal use of default interface methods.

algorithms presented in this paper, by contrast, the size overhead with maximal IM is at most a factor 1.8, and below a factor 1.6 for most benchmarks. While some of this reduction in size overhead is due to the profile-based OFI, the biggest responsible is the use of default Java methods, which is possible since Java 8.

Interestingly, even the size overhead reported in Figure 9, when no default interface methods are used but dummy class methods are used instead, compares rather well to the factors 6.2 and 5 mentioned above for the existing literature, i.e., our previous work [2]. While the numbers presented here are not smaller, they are more realistic because they were measured for dummy class methods with average sizes of 66 bytes. In the previous work, empty dummy methods were used instead. Obviously, such empty class methods are eye catchers for attackers, which would make custom attacks to determine the original class hierarchy trivial. The prototype implementation evaluated in this paper does not lack in that regard, and hence the results are much more representative of relevant uses of the presented obfuscations. The fact that the size overhead does not grow much compared to the previous results, despite using filled dummy class methods instead of empty ones, results from the improved MM algorithm as briefly discussed in Section 5 and from the more extensive deployment of IR: In the previous work, only method and field names were renamed and hence shortened, but no class and interface names.

Additionally, our new results show that the use of profile information for OFI results in more controllable, more consistent execution time overhead of the factories. For both the pre-existing prototype implementation reported in literature [2] and the new implementation evaluated in this paper, and for each set of obfuscation settings, we computed the standard deviation of the relative execution times over each of the ten obfuscated program versions that were generated with ten different random seeds per obfuscation setting. With the pre-existing algorithm implementation, the average and the maximum standard deviations computed over all benchmarks and obfuscations settings were rather large at 0.21 and 2.61, respectively. The large standard deviation of 2.61 was observed for xalan, for which two of the ten random seeds used to generate obfuscated program versions at maximum obfuscated settings resulted in program versions that executed significantly (6 times) slower than the other eight. With the algorithms and prototype presented in this paper, we did not encounter such large differences in execution time between versions generated using the same settings but with different random seeds: Both the average and the maximum standard deviations were much smaller, at 0.02 and 0.09, respectively. The deployment of the obfuscation techniques as described in this paper is hence much more predictable in terms of execution time overhead, and hence much more practical.

DOI: 10.1002/spe

In summary, our novel contributions result in overall size improvements controllable in a stealth-size trade-off, and in much more acceptable execution time overheads compared to the pre-existing state of the art. In addition, the performance overhead of program versions generated using the new techniques can be controlled better and is more consistent, as it depends less on the chosen random seed. It is therefore less likely for an obfuscator implementing the new techniques to generate an undesirable program version that performs significantly worse than other versions.

7.6. Obfuscation strength

Ideally, obfuscating transformations should be potent, resilient, and stealthy [22]. Being potent means they should make it more difficult for attackers to understand programs or to extract certain information from them. Being resilient means they should be difficult to undo or overcome by means of an automatic deobfuscator. Being stealthy means that it is hard to identify which protection has been applied, or precisely how it has been applied. In practice, measuring the level of protection provided by obfuscation techniques is a difficult task. Potency, resilience, and stealth are abstract terms that can only be made concrete by considering concrete attack methods that a protection mitigates. We know from practice and recent literature [23] that man-at-the-end attackers use a myriad of methods to reverse engineer obfuscated applications, ranging from completely manual to completely automated, from sound to unsound, from passive to active, from static to dynamic, etc. In their survey on the arms race between code obfuscation and code analysis, Schrittwieser et al. consider four classes of attack methods: pattern matching, automated static analysis, automated dynamic analysis, and human-assisted analysis [24]. The latter is used as a synonym for manual reverse engineering. In practice, multiple methods are often combined, e.g., by letting a tool analyze the code automatically, after which a human uses the analysis results to try to comprehend the code. We also know that protections need to be combined to protect against all possible attack methods, because protections deployed in isolation typically only have a minor effect on the so-called attack path of least resistance. It then follows that in order to demonstrate the practical strength of a proposed protection, one needs to study the extent to which it mitigates at least some (popular) attacks methods.

Doing so with empirical research is too expensive and time-consuming, however, so two alternative evaluation methodologies are commonly used instead: measuring how the efficiency and effectiveness of commonly used attack tools is affected by the protection, and approximating attack complexity by means of software complexity metrics from the domain of software engineering [22, 25, 6]. These evaluation methodologies complement each other nicely, as the former focuses on the use of automated analysis tools to overcome protections, while the latter relates to manual tasks in the reverse engineering process. So mitigation against both automated and manual attack methods can be evaluated, thus covering both potency and resilience. In this work, we evaluate the proposed protection with both methodologies.

7.6.1. Mitigation of automated static analysis To measure the complexity of transformed applications from the perspective of an automated static analysis tool, we used the average points-to set size as a metric. Large values for this metric imply that the results of other automated reverse engineering processes such as virtual call resolution and call graph construction become less precise. Furthermore, additional analysis techniques that rely on this information, such as program slicing [26, 27], also become less precise and often also slower and more memory-intensive because more information needs to be tracked and propagated. Points-to set sizes are thus relevant in terms of potency, because they provide an indication about the precision with which analysis tools will be able to present information to an attacker. Points-to set sizes are also relevant with respect to resilience, as smaller points-to set sizes enable more automatic refactoring, including refactorings that would undo the obfuscation by converting the obfuscated hierarchy back in the direction of the original one. For a more detailed description of how points-to sets relate to refactoring possibilities, we refer to the existing literature on refactoring using type constraints [18].

Figure 11 shows the average points-to set sizes of the different benchmark versions, obtained using WALA's 01-container-CFA analysis, the most precise pointer analysis offered by this popular

DOI: 10.1002/spe

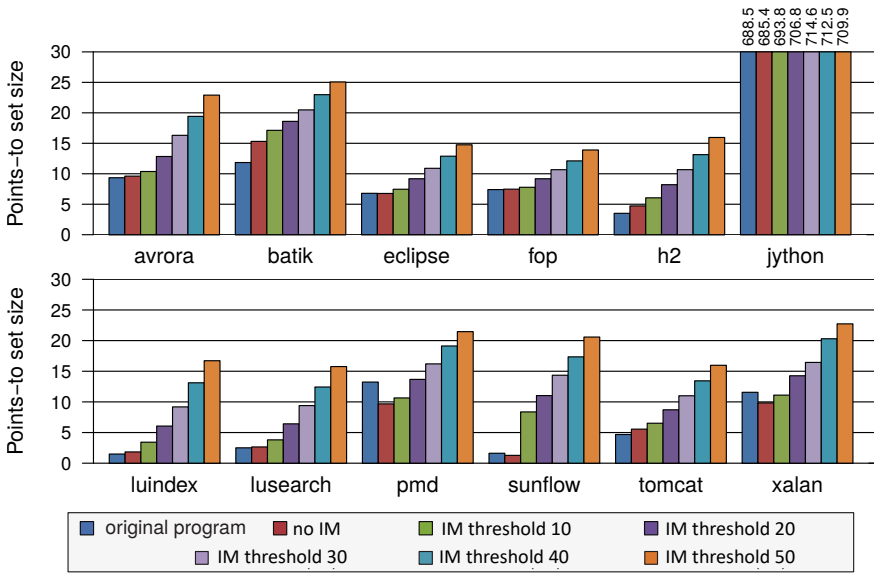

Figure 11. Average points-to set sizes of original and obfuscated benchmark versions with maximal use of default interface methods.

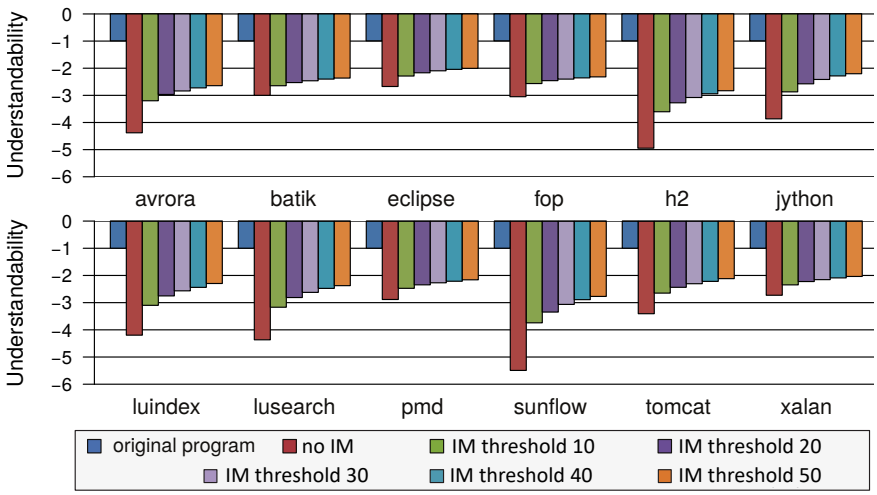

Figure 12. Average QMOOD understandability scores for original and obfuscated benchmark versions with maximal use of default interface methods.

and powerful Java bytecode analysis tool [16]. The chart shows that the techniques presented in this paper impact the points-to set sizes significantly in the direction we want, with more IM leading to larger points-to sets. Compared to our previous work [2], the points-to set sizes of the obfuscated versions appear to have even grown. This is due to the use of an updated version of the WALA analysis tool, as needed to handle Java 8 bytecode. When we recompute points-to set size with the updated WALA on program versions obfuscated with the prototype obfuscator for Java 6 code used in our earlier work, we get the same results.

Finally, we note that the points-to set sizes do not depend on the use of default interface methods or class methods to implement dummy methods. Since their bodies do not contain real code (i.e., they exist of nops) in our current prototype, they do not have any points-to sets. If a future, feature-complete industrial tool fills their bodies with code similar to that found in other methods (as needed

DOI: 10.1002/spe

to improve the stealthiness and to counter matcher-based attacks), the average points-to sets in them will also be similar to those already present, so the overall numbers will not change.

We conclude that when all the proposed transformations are combined, including sufficiently aggressive IM, points-to sets computed with state-of-the-art algorithms grow significantly. They grow with increasing IM, and are as effective as the previous state of the art. Our techniques therefor clearly provide real protection against the many automated analysis for which attackers rely on points-to sets to build as precise as possible understandings of the software under attack to improve and speed-up their reverse engineering attacks.

7.6.2. Mitigation of manual reverse engineering To measure the complexity of protected programs from the perspective of a human attacker that manually (i.e., mentally) tries to comprehend the code, we should preferably use a validated complexity metric from the domain of software engineering. Furthermore, the metric needs to be relevant for the type of obfuscations we propose, which amount to design obfuscation. Finally, the metric needs to fit the type of programming language we target, so it needs to fit object-oriented languages. It hence does not make sense, e.g., to use the well-known metric of cyclomatic numbers that measures the complexity of the control flow in function bodies [28], simply because our obfuscations do not at all target control flow. With respect to data flow, our obfuscations do not target or alter the true data flow, they only lower the precision of static data flow analysis by obfuscating the type information needed to compute precise points-to sets on which data flow analyses depend. Since we already measured points-to set sizes, there is little value in adding more data flow related complexity metrics such as those based on information flow [29].

After searching the literature for possible candidates, we settled for the *understandability* metric formalized by Bansiya and Davis in their Quality Model for Object-Oriented Design (QMOOD) [30]. When applied to two versions of an application, this software-complexity metric expresses whether the perceived complexity of version 1 is higher or lower than that of version 2. The value for this conceptual metric of understandability is to be computed as a linear combination of seven concrete metrics that each measure a different aspect of a program's design by counting syntactic features of the source code or the bytecode, which is equivalent in the case of Java when no anti-decompilation obfuscations are deployed [6]. The concrete metrics (and their weights) are abstraction (-0.33), encapsulation (0.33), coupling (-0.33), cohesion (0.33), polymorphism (-0.33), complexity (-0.33), and design size (-0.33) [30]. These metrics are measured on the baseline version and on a second version, and their values relative to the baseline values are summed up with the chosen weights. So by definition, the baseline version of each benchmark has an understandability score of -0.99. More negative values correspond to less understandable versions, as negative weights indicate that a higher metric value implies less understandability.

It is important to note that QMOOD understandability is a first-order proxy for the real program understandability: It is a metric computed directly on a syntactical, structural representation of the software, i.e., the abstract syntax trees that represent the whole code base. It therefore only captures syntactical features of the code, and neglects other features such as data flow complexity. It hence complements the metric of points-to set sizes, rather than providing a confirmation or refutation of the results discussed above.

Figure 12 presents an overview of the understandability scores for each of the benchmark versions when all dummy methods are implemented by means of default interface methods. It can be seen that without IM, the understandability of the obfuscated benchmarks ranges from -2.6. to -5.5. As more interfaces get merged, the understandability improves again. When up to 50 interfaces get merged together, the understandability varies between -2.0 and -2.8. To understand why understandability decreases with CHF and IOF, and increases when more interfaces are merged, it is useful to break down understandability into the contributing metrics in isolation. Figure 13 depicts this breakdown for the lusearch benchmark. This benchmark is representative for all of the benchmarks, as all of them show similar trends as a result of our obfuscating transformations. It are the metric values charted in this figure that are combined, using the aforementioned weights of -0.33 and +0.33, to compute the QMOOD understandability metric. In the case of Java, the metrics are computed on

DOI: 10.1002/spe

classes and interfaces, as if interfaces are abstract classes. So in the description below, the term "classes" means "classes and interfaces".

- *Abstraction* This metric measures the average number of ancestors per class in a class hierarchy. CHF by construction reduces this metric. The interfaces merged during IM all have 0 ancestors (besides the not counted java.lang.Object). This means that the classes with the lowest number of ancestors are removed from the mix of all classes. The average number of ancestors per remaining class therefore increases when more IM is applied.

- *Encapsulation* Over all classes in a program, this metrics measures the ratio beween private and protected fields on the one hand and all fields on the other hand. Fields in ancestor classes in the original hierarchy are duplicated into their descendants during CHF, i.e., from classes higher in the original hierarchy to classes lower into that hierarchy. As there are relatively more private and protected fields in classes higher in the original hierarchy, CHF increases the metric. IM does not affect it.

- *Coupling* Averaged over all classes, this metric measures per class how many other classes are referenced in class' field declarations and method signatures. MM as part of IM injects references to new interfaces into method signatures when it merges parameter lists, without necessarily replacing all original references. With limited IM and hence limited MM, the coupling metric increases as a result. When IM and MM are applied more aggressively, however, the level of coupling saturates: More references to classes are still added in the signatures of more heavily merged methods, but the additionally referenced classes typically already were mentioned somewhere else in the merged methods' classes anyway. So the metric no longer increases that much because of increasing MM. Moreover, because the number of different interfaces decreases with more IM, the number of different interfaces referenced also decreases with more IM. For higher levels of IM, this makes the coupling metric decrease again.

- *Cohesion* Averaged over all classes, this metric measures the extent to which a class' method signatures contain the same classes. CHF reduces this because methods get duplicated from ancestor to descendant classes. The flattened classes then contain methods originating from multiple original classes. As there was less cohesion between those original classes than within them, the cohesion of the flattened hierarchy is also lower. IM and can impact this metric in both ways: Some merging operations of specific methods decrease the overlap between the references in their signatures with those in other methods' signatures, some merging operations increase it. On average, IM and MM have little impact on this metric, which is therefore flat in the chart for varying levels of IM.

- *Polymorphism* During CHF, methods are duplicated and references to original classes are replaced by references to interfaces. The number of methods calls that are polymorphic hence explodes when CHF is applied. When interfaces and methods get merged, however, and dummy methods are implemented as default interface methods, the number of polymorphic calls decreases. Hence the chart shows declining polymorphism bars.

- *Complexity* This metric simply counts the number of method definitions in the whole program. CHF obviously adds many of them by duplicating methods. IM and MM then again decrease the number of method definitions, in particular when dummy methods are implemented as default interface methods.

- *Design Size* This metric simply counts the number of classes in the program. CHF and OFI increase this by adding interfaces and factory classes. IM reduces the number of interfaces, so more IM implies a decreasing metric.

As more IM and correspondingly more MM reduce four of the five metrics with negative weights in the formula for understandability, it is not surprising that understandability increases when they are applied more aggressively.

DOI: 10.1002/spe

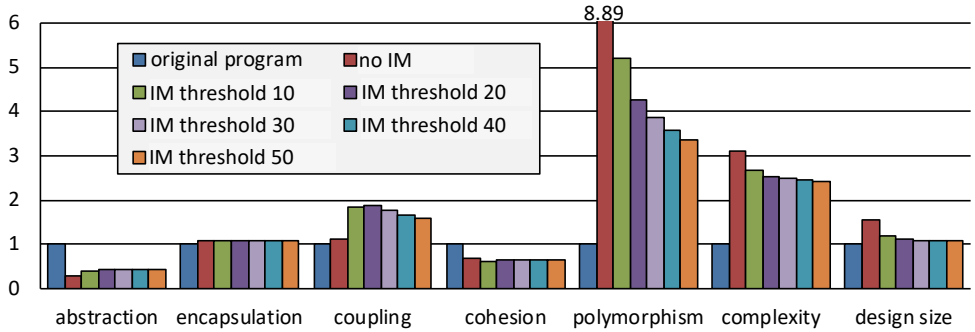

Figure 13. QMOOD metrics breakdown for the lusearch benchmark when dummy methods are implemented maximally as default interface methods.

Figure 14 shows the QMOOD understandability results obtained for benchmark versions in which dummy methods are implemented as class methods. Figure 15 breaks down the understandability for the lusearch benchmark with class dummy methods. The latter chart differs in two significant ways from the one in Figure 13 for the version with default interface methods. First, the complexity metric scales differently. When class dummy methods are used, more IM directly implies that the number of dummy methods grows, because classes then implement all methods in the merged interfaces they implement. MM limits their number, but not enough to prevent that the number of methods keeps growing. Similarly, the coupling metric grows much more than when default interface methods are used. The reason is similar: with dummy class methods, the classes contain many merged dummy methods, which reference a large number of other classes in their signatures simply because they are the result of merging dummy methods with different classes in their signatures. So the average number of classes referenced by each class in method signatures increases with higher levels of IM and the corresponding higher levels of MM. As a result of the different scaling of the complexity and coupling metrics, the QMOOD understandability evolves differently: for dummy methods implemented as class methods, Figure 14 shows that the understandability decreases with increasing amounts of IM, rather than increasing as was the case with default interface methods. This is in fact no surprise: The obtained results in understandability for benchmark versions with dummy class methods resemble those published in earlier work [2], as for that earlier obfuscator also implemented all dummy methods as class methods.

We conclude that our proposed obfuscations effectively reduce the understandability of the software as measured with the QMOOD metric. With increasing IM (and hence MM), the understandability decreases or increases depending on the use of class dummy methods versus default interface methods. On top of the possibility to counter matcher-based attacks as discussed in Section 4, this is another indication that the injection of class dummy methods is a worthwhile consideration when deploying our proposed combination of protections. When class dummy methods are deployed maximally, the negative impact of our transformations on QMOOD understandability is comparable to that of the pre-existing state of the art.

The overall conclusion of the obfuscation strength evaluation is that our transformations effectively provide protection against a number of attack methods, including at least automated static analysis and human code comprehension, and that users of the obfuscation tool can configure it to trade-off the protections' effectiveness against different classes of such attack methods as well as their efficiency, i.e., their overhead.

8. RELATED WORK

The combination of IR, CHF, IM with MM, and OFI is not novel, we already proposed to combine them in our previous work [2]. Compared to the original proposal and evaluation thereof, the main differences are that (1) we added support to use default interface methods as available since Java 8;

DOI: 10.1002/spe

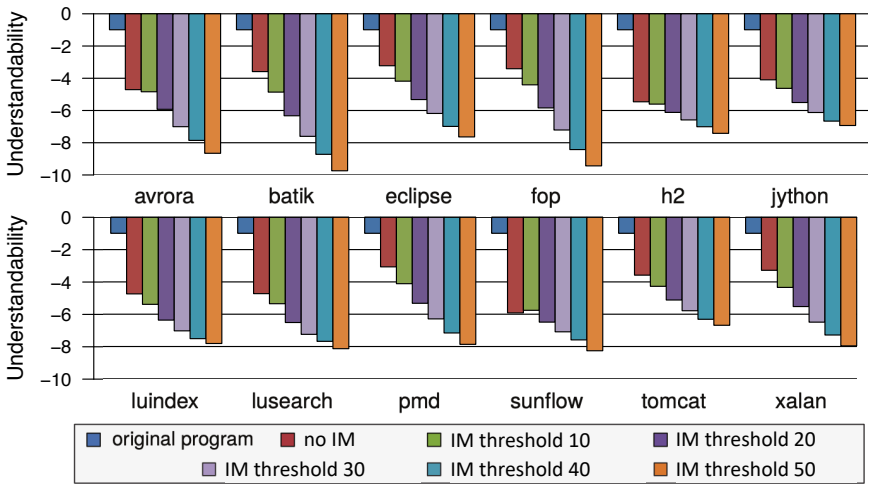

Figure 14. Average QMOOD understandability scores for original and obfuscated benchmark versions, with dummy class methods.

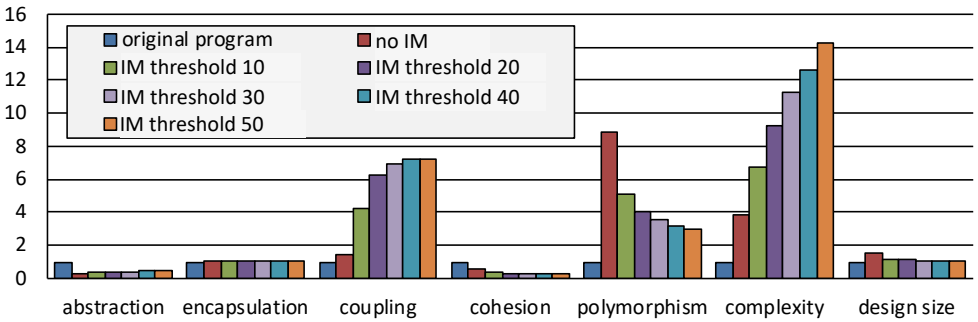

Figure 15. QMOOD metrics breakdown for the lusearch benchmark when dummy methods are implemented as class methods.

(2) we slightly improved MM; (3) we presented a completely redesigned OFI which is now based on profile information; (4) we now apply IR more aggressively by also renaming class names; (4) we now generate dummy method bodies of realistic sizes. The main result is that we obtain a comparable level of protection effectiveness against reverse engineering, but we do so much more efficiently, i.e., with a radically reduced overhead in terms of class file size growth and execution slowdown. Moreover, the overhead is now more predictable. This eases the efficiency-effectiveness trade-off and implies that less application-specific configuration tuning will typically be needed than with the original work.

The way MM combines methods is related to the generic *method interleaving* transformation by Collberg, Thomborson and Low [22]. Method interleaving merges two methods, creating a new method whose body consists of the bodies of the original methods guarded by conditional checks. The parameter type list of the merged method consists of the combined parameter type lists of the original methods, and an additional parameter to determine which of the original method bodies needs to be executed. The Sandmark tool implements that form of method interleaving for Java [31], and in that tool, it is also named MM. That Sandmark version of MM differs significantly from the MM method we use, however. Sandmark's version of MM only merges static methods by combining their signatures, by combining their bodies, and by adding another (int) parameter on which the combined bodies are multiplexed. The latter is similar to the way our object factories get additional parameters on which we multiplex invocations of the original constructors.

DOI: 10.1002/spe

Sandmark's MM is applied as the sixth and last pass in a algorithm called Method 2R Madness Obfuscation by Martin Stepp and Kelly Heffner [32]. The preceding passes create the static methods by lifting them from existing dynamic methods, and alter the signatures to prepare for the merging. As such, MM in Sandmark is actually also used to merge bodies that originate from dynamic bodies, and conceptually it is hence also applied to dynamic methods, just like our MM can be applied to dynamic methods. Our MM involves no additional parameters and multiplexing, however, as we only merge methods of which at least one is a dummy method in each class and interface implementing the methods. This difference coincides with the fact that MM in Sandmark is deployed as a design obfuscation in itself, which aims to obfuscate the separation of functionality into methods, whereas our MM servers the purpose to reduce the class file size and to improve the resilience against custom attacks specifically targeting the obfuscation achieved with CHF and MM.

An interesting difference between our obfuscation and Method 2R Madness is that merged methods in the latter have signatures comprising only one Object[] parameter and one multiplexing parameter of type int. At first sight, Method 2R Madness hence seems to hide more type information. However, wherever elements from that Object[] parameter are used in the multiplexed method bodies, they are first cast to their original type in order to remain valid statically typed code. Also, all calls to the original dynamic methods remain present in the (multiplexed) code. Those dynamic methods remain present in the classes in an unchanged hierarchy and with their original signatures (unlike in our obfuscation); those methods have simply become wrappers that invoke the merged static methods, passing their arguments with a value for the multiplexing. Given the presence of the original types in the dynamic method signatures, the presence of all the casts in the merged static methods, and the unchanged class hierarchy, we do not think Method 2R Madness' feature of only having an Object[] parameter actually hides much useful type information in the presence of type inference algorithms. A detailed study of that aspect of Method 2R Madness is left for future work.

The Java Binary Enhancement Tool (JBET) developed as part of the Self-Protecting Mobile Agents (SPMA) project by Badger *et al.* [33, 34] implements a more aggressive form of method interleaving. As part of its obfuscation routine JBET first flattens the control flow [35] of all methods, before merging them into a single flattened method to hide the boundaries between the original methods.

To reduce code size effectively, MM is driven by a model that estimates the change in application size as a result of applying a merge operation, as described in detail in Foket's PhD thesis [9]. Neither Collberg *et al.*, nor Badger *et al.* describe a way to measure the impact of their techniques on the resulting size of an application. In fact, we are unaware of any transformation described in literature that is able to model the effects of structural operations such as method merges on the size of the application being transformed.

Our proposed form of OFI bears some resemblance to Sakabe, Soshi, and Miyaji's type hiding obfuscation [36]. The main difference between the code generated by our and their approaches is that our approach generates factory methods, rather than inline code to create new objects. This enables us to generate larger, more complex factories without increasing the size of the application too much.

To reduce or limit the execution time overhead of transformed programs, program transformation frameworks such as optimizing compilers [37], link-time optimizers [38], and obfuscators often make use of execution profile information. Linn and Debray, for example, use profile information to limit the overhead of branch functions [39], while Popov *et al.* use it to limit the overhead of their obfuscation that abuses signal handling [40].

In both of these examples, and in most other profile-driven techniques, decisions whether or not to transform a code fragment are based on whether or not its (relative) execution frequency is below a specified threshold. Such approaches come with two drawbacks. Firstly, it may be difficult to manage the overhead in a fine-grained way, because there is no global overhead budget, and the decision of whether or not to transform a code fragment is made independently for each code fragment. As a result, the overhead can still be large, even for low thresholds. Secondly, depending on their execution profiles, it may be necessary to select a different threshold for different applications, which is not desirable.

DOI: 10.1002/spe

In comparison, OFI uses a more fine-grained approach in which the decision on whether or not to perform a specific merge operation is based on how much is left of a global overhead budget. This enables us to manage its overhead more carefully. Furthermore, because the overhead budget expresses how much the maximum number of dynamic arguments used to create objects is allowed to increase, we can obtain similar levels of overhead for multiple applications using the same threshold, even if the applications have different execution profiles.

Contrary to our OFI transformation, the type hiding transformation by Sakabe *et al.* does not make use of profile information to decide where to insert inline factories. One may argue that this is not necessary, given that the average increase in execution time of their transformation (in combination with two other obfuscating transformations they present[‖]) is only 43%. However, the applications they used for their evaluation are more than two orders of magnitude smaller (in number of classes) than the ones we used during our evaluation. In this respect, it would be interesting to perform a comparative study of their and our techniques on larger benchmarks like ours, especially given that their technique does not generate factory methods, and hence does not require dummy arguments to be passed between methods to create a new object.

9. CONCLUSION

We presented a combination of five transformations to obfuscate type information in Java programs: identifier renaming, class hierarchy flattening, interface merging with method merging, and object factory insertion. That combination in itself is not new, but we improved the algorithms and their deployment to reach the same level of obfuscation effectiveness of previous work at a much higher efficiency, i.e., with much less overhead. Compared to the work presented before in [2], we applied identifier renaming to more identifiers, we made the introduction of so-called dummy class methods during class hierarchy flattening optional by exploiting Java 8 default interface methods, we used a refined decision logic for choosing candidates for method merging, we developed a completely new object factory insertion algorithm that exploits profile information, and our prototype tool now generates realistically sized dummy methods. As a result of the latter, some custom reverse-engineering attack methods that had been neglected before, such as the extraction of information about related classes from the sizes of potential dummy methods, are now mitigated.

An evaluation on the DaCapo benchmark suite showed that the new algorithms and implementation result in comparable levels of protection as the previous work [2] in terms of points-to set sizes (which become up to 10 times larger for some benchmarks) and human understandability (which becomes up to 10 times harder for some benchmarks as measured with the QMOOD metric), two metrics relevant to evaluate obfuscation's potency and resilience with respect to reverse-engineering attacks. No benchmark slowed down more than 70%, with most benchmarks slowing down only about 20%; and for all benchmarks the class file size overhead could be kept under 90%. Compared to the previous state of the art [2], the worst-case execution time overhead and the worst-case application size overhead were hence reduced with factors 4/5 and 2/3, respectively.

The obfuscation algorithms can further be parameterized to trade off the effectiveness and efficiency, i.e., the achieved level of protection and the resulting overhead. So for a developer with a limited budget in terms of performance and code size overhead, the contributions of this paper imply that the developer can now obtain stronger protection than what was possible before.

The main result and value of the presented work is hence that it makes the overhead of the achieved obfuscation potentially acceptable in many industrial and commercial deployment scenarios, thus making the real-world deployment of the techniques much more feasible, whereas it was infeasible because of excessive costs before this work.

Interested practitioners or researchers can try out and experiment with our prototype implementation, which is available under a liberal open-source license at `http://gujto.elis.ugent.be/`. All necessary scripts to reproduce our results are also available there.

[‖] The authors only report overheads for programs on which all three transformations are applied.

DOI: 10.1002/spe

As future work, we see a potential for speeding up the obfuscator algorithms, and for alternative IM strategies targeted at maximizing the effectiveness of OFI. Such strategies could, e.g., try to estimate the effect of merging different interface combinations on points-to set sizes. Furthermore, the larger points-to sets and call graphs obtained after our obfuscations open up opportunities for alias-based program obfuscations.

ACKNOWLEDGEMENTS

This work was supported by the Agency for Innovation by Science and Technology in Flanders (IWT), and by Ghent University. The authors would like to thank Ghent University, the Flemish Supercomputer Center (VSC), the Hercules Foundation and the Flemish Government - department EWI for the STEVIN Supercomputer Infrastructure on which we carried out part of this work. The authors also thank Sofie, Sándor, and Adriaan for providing useful feedback on nightly drafts of this paper.

REFERENCES

1. Collberg C, Nagra J. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.
2. Foket C, De Sutter B, Koen DB. Pushing java type obfuscation to the limit. *IEEE Transactions on Dependable and Secure Computing* Nov 2014; **11**(6):553–567.
3. Blackburn SM, *et al.*. Wake up and smell the coffee: evaluation methodology for the 21st century. *Communications of the ACM* Aug 2008; **51**(8):83–89.
4. Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot - a Java bytecode optimization framework. *Proc. Conference of the Centre for Advanced Studies on Collaborative Research*, 1999; 125–135.
5. Lam P, Bodden E, Lhoták O, Hendren L. The Soot framework for Java program analysis: a retrospective. *Cetus Users and Compiler Infrastructure Workshop*, 2011.
6. Batchelder M, Hendren L. Obfuscating Java: the most pain for the least gain. *Proc. International Conference on Compiler Construction*, 2007; 96–110.
7. Coppens B, De Sutter B, Maebe J. Feedback-driven binary code diversification. *ACM Trans. Archit. Code Optim.* Jan 2013; **9**(4):24:1–24:26.
8. Coppens B, De Sutter B, De Bosschere K. Protecting your software updates. *IEEE Security Privacy* March-April 2013; **11**(2):47–54.
9. Foket C. Global obfuscation of bytecode applications. PhD Thesis, Ghent University 2015.
10. Bellamy B, Avgustinov P, de Moor O, Sereni D. Efficient local type inference. *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2008; 475–492.
11. Collberg C, Thomborson C, Low D. Manufacturing cheap, resilient, and stealthy opaque constructs. *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998; 184–196.
12. Zhou Y, Main A, Gu YX, Johnson H. Information hiding in software with mixed boolean-arithmetic transforms. *WISA'07: Proceedings of the 8th international conference on information security applications*, 2007; 61–75.
13. Gagnon E, Hendren L, Marceau G. Efficient inference of static types for Java bytecode. *Proc. International Symposium on Static Analysis*. Springer, 2000; 199–219.
14. Bodden E, Sewe A, Sinschek J, Mezini M. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. *Proceedings of the 33rd International Conference on Software Engineering*, 2011; 241–250.
15. Collberg C, Myles G, Stepp M. An empirical study of java bytecode programs. *Softw. Pract. Exper.* May 2007; **37**(6):581–641, doi:10.1002/spe.v37:6. URL `http://dx.doi.org/10.1002/spe.v37:6`.
16. Dolby J, Fink SJ, Sridharan M. TJ Watson libraries for analysis (WALA). `http://wala.sf.net` 2012.
17. Fowler M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley: Boston, MA, USA, 1999.
18. Tip F, Furher R, Kiezun A, Ernst M, Balaban I, De Sutter B. Refactoring using type constraints. *ACM Transactions on Programming Languages and Systems* 2011; **33**(3):9:1–9:47.
19. Streckenbach M, Snelting G. Refactoring class hierarchies with KABA. *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004; 315–330.
20. Gong L. *Java™SE Platform Security Architecture*. [Online] Available: http://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/ security-spec.doc.html.
21. Kalibera T, Jones R. Rigorous benchmarking in reasonable time. *Proceedings of the 2013 International Symposium on Memory Management*, 2013; 63–74.
22. Collberg C, Thomborson C, Douglas L. A taxonomy of obfuscating transformations. *Technical Report*, University of Auckland 1997.
23. Ceccato M, Tonella P, Basile C, Falcarin P, Torchiano M, Coppens B, De Sutter B. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. *EMPIRICAL SOFTWARE ENGINEERING* 2018; **24**(1):240–286. URL `http://dx.doi.org/10.1007/s10664-018-9625-6`.
24. Schrittwieser S, Katzenbeisser S, Kinder J, Merzdovnik G, Weippl E. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.* Apr 2016; **49**(1):4:1–4:37.
25. Anckaert B, Madou M, De Sutter B, De Bus B, De Bosschere K, Preneel B. Program obfuscation: A quantitative approach. *Proceedings of the 2007 ACM Workshop on Quality of Protection*, QoP '07, ACM: New York, NY, USA, 2007; 15–20.
26. Weiser M. Program slicing. *Proc. 5th Int'l Conference on Software Engineering*, 1981; 439–449.

27. Tip F. A survey of program slicing techniques. *Journal of programming languages* 1995; **3**(3):121–189.
28. McCabe TJ. A complexity measure. *IEEE Transactions on Software Engineering* Dec 1976; **SE-2**(4):308–320, doi:10.1109/TSE.1976.233837.
29. Henry S, Kafura D. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering* Sep 1981; **SE-7**(5):510–518, doi:10.1109/TSE.1981.231113.
30. Bansiya J, Davis CG. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering* Jan 2002; **28**(1):4–17.
31. Collberg C, Myles GR, Huntwork A. Sandmark-a tool for software protection research. *IEEE Security Privacy* July 2003; **1**(4):40–49, doi:10.1109/MSECP.2003.1219058.
32. Collberg C, Miklofsky M, Myles G, Purushotham A, Rajendran R, Huntwork A, Zhang X, Mandel D, Segurson A, Stepp M, *et al.*. Sandmark algorithms. *Technical Report*, University of Arizona 2002.
33. Badger L, D'Anna L, Kilpatrick D, Matt B, Reisse A, Vleck TV. Self-protecting mobile agents obfuscation techniques evaluation report. *Technical Report 01-036*, NAI Labs Nov 2001.
34. D'Anna L, Matt B, Reisse A, Vleck TV, Schwab S, LeBlanc P. Self-protecting mobile agents obfuscation report — Final report. *Technical Report 03-015*, Network Associates Laboratories Jun 2003.
35. Wang C, Hill J, Knight J, Davidson J. Protection of software-based survivability mechanisms. *dsn* 2001; **00**:0193.
36. Sakabe Y, Soshi M, Miyaji A. Java obfuscation approaches to construct tamper-resistant object-oriented programs. *IPSJ Digital Courier* 2005; **1**:349–361.
37. Gupta R, Mehofer E, Zhang Y. Profile guided compiler optimizations 2002; .
38. De Bus B, De Sutter B, Van Put L, Chanet D, De Bosschere K. Link-time optimization of arm binaries. *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2004; 211–220.
39. Linn C, Debray S. Obfuscation of executable code to improve resistance to static disassembly. *Proc. 10th ACM Conference on Computer and Communications Security*, 2003; 290–299.
40. Popov IV, Debray SK, Andrews GR. Binary obfuscation using signals. *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007; 19:1–19:16.

DOI: 10.1002/spe