A Novel Obfuscation: Class Hierarchy Flattening

Christophe Foket*, Bjorn De Sutter, Bart Coppens, and Koen De Bosschere

Computer Systems Lab
Electronics and Information Systems Department
Ghent University
Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium
{cfoket,bcoppens,kdb,brdsutte}@elis.ugent.be

Abstract. This paper presents class hierarchy flattening, a novel obfuscation technique for programs written in object-oriented, managed programming languages. Class hierarchy flattening strives for maximally removing the inheritance relations from object-oriented programs, thus hiding the overall design of the program from reverse engineers and other attackers. We evaluate the potential of class hierarchy flattening by means of a fully automated prototype tool for Java bytecode. For real-life programs from the DaCapo benchmark suite, we demonstrate that the transformation effectively hinders both human and tool analyses, and that it does so at limited overheads.

Key words: Java bytecode, obfuscation, class hierarchy, program design

1 Introduction

Reverse engineering and modification of managed code are well-understood and common practices, with many legitimate goals [16]. Malicious developers can abuse them, however, to attack Java and .NET applications with the goals of software piracy, software IP theft, and data theft. Their attacks are facilitated by the fact that managed code is executed at a high abstraction level. To combine run-time efficiency with programmer productivity, a large amount of symbolic information needs to be presented to the virtual machines that execute the code. This is needed, e.g., to enable effective and efficient just-in-time (JIT) compilation, to support efficient garbage collection, and to support reflection and bytecode verification. This symbolic information is also what makes managed code easier to understand, reverse engineer, decompile, modify, reuse and steal.

With respect to reverse engineering (and all practices for which reverse engineering is a prerequisite), many different obfuscation techniques have been proposed. Some try to prevent automatic decompilation [5, 17], some try to hide

* The authors want to thank the Agency for Innovation by Science and Technology in Flanders (IWT) for their support and Ghent University, the Hercules Foundation and the Flemish Government - department EWI who funded the STEVIN Supercomputer Infrastructure at Ghent University on which we carried out part of this work.

data (flow) properties [8, 34] or control flow properties [8, 9, 17, 19, 20, 22, 24, 33] from both tools and humans. Others try to remove information that is useful informally, such as field and method identifiers [2, 5]. Finally, a few have proposed obfuscating the overall application design by altering the class and interface hierarchy to make it harder to understand for software engineers [28]. The latter techniques aim for the opposite of classic code refactoring [27, 31].

This paper takes application design obfuscation one step further. Instead of merely modifying an application's type hierarchy, we propose a technique called class hierarchy flattening (CHF) to get rid of it altogether. Given a number of constraints because of, e.g., compatibility with external libraries, CHF strives for a class hierarchy that is as flat as possible, i.e., in which application classes are siblings rather than subtypes and supertypes. We discuss the necessary analyses and transformations to automate CHF and present a proof-of-concept tool. We evaluate the level of software protection provided by CHF and its overhead.

The remainder of this paper is structured as follows. First, Section 2 discusses the conceptual goals of CHF by means of an example program. The transformation itself is discussed in some detail in Section 3, and evaluated in Section 4. We compare CHF to related work in Section 5. Finally, Section 6 draws conclusions and discusses some future extensions.

2 Rationale: an Example Program

To set the context for a detailed discussion of our obfuscating class hierarchy transformation, we first present an example consisting of a media player. It consists of three main parts: (1) the player initializer (2) support for media files and (3) support for media streams contained in the media files. Different subtrees of the class hierarchy implement those parts, as shown in Figure 1. The code in Figure 2(a) illustrates their interaction.

The `main` method of class `Player` creates an array of `MediaFile` objects to be played (line 10). It then queries each of the media files in this list for its media streams (line 12), which are initialized when the media file is accessed with the `readFile` method. Figure 2(a) shows how this is done for the `MP3File` class, which represents MP3 files containing MPEG audio streams.

During playback, the player checks the run-time type of the `MediaStream` object associated with the stream (lines 13 and 15) to decide where it needs to be output. Depending on the actual run-time type of the `MediaStream` objects, they are either cast to `AudioStream` or `VideoStream`, such that the correct `play` method is invoked (lines 14 and 16). The `play` methods essentially output the raw bytes of the media streams' analog signals for a specific output device. Those bytes are obtained, decrypted (lines 35–36) and decoded (line 37) with the `getRawBytes` method declared in `MediaStream`. Note that because the decoding process is different for each type of media stream, the `decode` method is declared as abstract, such that it can be implemented by subclasses of `MediaStream`. The decryption process, on the other hand, is the same for each type of media stream and is therefore handled by the `MediaStream` class.

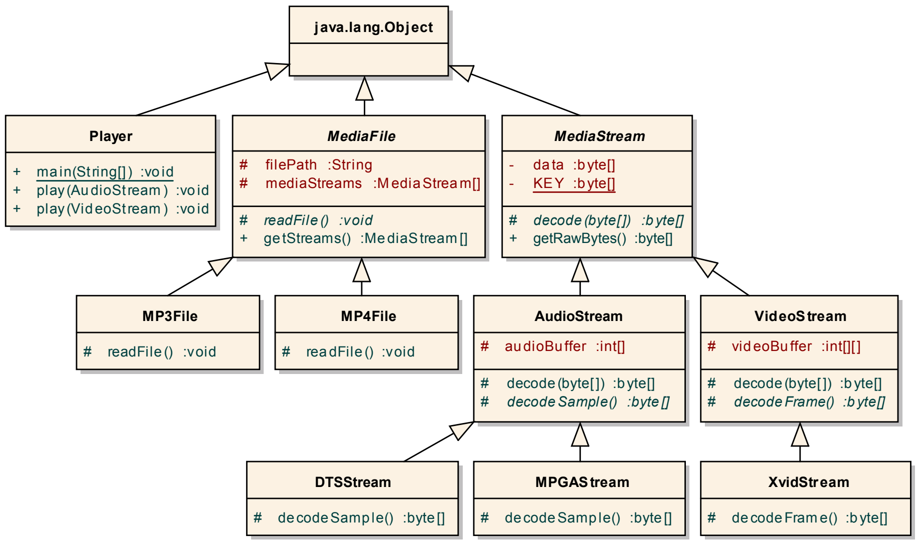


Fig. 1: Class hierarchy of a simple DRM media player

From a software-engineering point of view, the media player application is well structured. The inheritance relations are meaningful and code shared between different classes is located in a common superclass. While we could have further improved the structure of the program by factoring out the casts and run-time type checks, we chose not to do so for educational purposes.

From a security perspective, however, some problems arise. First, the class hierarchy provides reverse engineers with information about the relationships between classes and the abstraction levels of the functionalities provided by classes (with classes higher in the hierarchy typically providing more abstract functionality). Secondly, code sharing through inheritance enables attacks in which compromising one class can cause all of its subclasses to be compromised. All media streams are decrypted using the `getRawBytes` method declared in `MediaStream`. Therefore, when an attacker reverse-engineers this method, he will be able to decrypt all supported media stream types. Finally, we observe that the program contains much type information, which simplifies both manual analysis and automated analysis that rely on, a.o., point-to set computations.

These issues can be solved by rewriting the well-structured hierarchy into the unstructured class collection of Figure 3. To determine how classes are related, an attacker can then no longer rely on the class hierarchy. He will instead have to analyze and compare all classes in the application. Furthermore, as all classes are provided with a (diversified) copy of all fields and methods declared in their former superclasses, they have become functionally more independent. Code is no longer shared between related classes, so attackers can no longer attack many classes at once by patching their common superclass. In short, more actual code analysis and tampering will be needed to mount a successful attack.

```
public class Player{
  public void play(AudioStream as) {
    /* send as.getRawBytes() to audio device */
  }
  public void play(VideoStream vs) {
    /* send vs.getRawBytes() to video device */
  }
  public static void main(String[] args) {
    Player player = new Player();
    MediaFile[] mediaFiles = ...;
    for(MediaFile mf : mediaFiles) {
      for(MediaStream ms : mf.getStreams())
        if(ms instanceof AudioStream)
          player.play((AudioStream)ms);
        else if(ms instanceof VideoStream)
          player.play((VideoStream)ms);
    }
  }
}

public class MP3File extends MediaFile {
  protected void readFile() {
    InputStream inputStream = ...;
    byte[] audioData = new byte[...];
    inputStream.read(audioData);
    AudioStream as = new MPGAStream(audioData);
    mediaStreams = new MediaStream[]{as};
  }
}

public abstract class MediaStream {
  public static final byte[] KEY = ...;
  public byte[] getRawBytes() {
    byte[] decrypted = new byte[data.length];
    for(int i = 0; i < data.length; i++)
      decrypted[i] = data[i] ^ KEY[i];
    return decode(decrypted);
  }
  protected abstract byte[] decode(byte[] data);
}
```

(a) original code

```
public class Player implements Common {
  public void play(Common as) {
    /* send as.getRawBytes() to audio device */
  }
  public void play1(Common vs) {
    /* send vs.getRawBytes() to video device */
  }
  public static void main(String[] args) {
    Common player = new Player();
    Common [] mediaFiles = ...;
    for(Common mf : mediaFiles) {
      for(Common ms : mf.getStreams())
        if(myChecker.isInstance(0, ms.getClass()))
          player.play(ms);
        else if(myChecker.isInstance(1, ms.getClass()))
          player.play1(ms);
    }
  }
}

public class MP3File implements Common {
  public void readFile() {
    InputStream inputStream = ...;
    byte[] audioData = new byte[...];
    inputStream.read(audioData);
    Common as = new MPGAStream(audioData);
    mediaStreams = new Common []{as};
  }
}

public class MediaStream implements Common {
  public static final byte[] KEY = ...;
  public byte[] getRawBytes() {
    byte[] decrypted = new byte[data.length];
    for(int i = 0; i < data.length; i++)
      decrypted[i] = data[i] ^ KEY[i];
    return decode(decrypted);
  }
  public abstract byte[] decode(byte[] data);
}
```

(b) flattened code

Fig. 2: Partial implementation of the `Player`, `MediaStream` and `MP3File` classes

Code analysis has also become harder, as the code in the transformed application (shown in Figure 2(b)) contains less type information than the original application. This follows from all declaration types being replaced by a new `Common` type. Since this interface type serves as a common supertype for all classes in the application and declares all their instance methods, all classes implement a significantly larger number of methods. The subset of those methods that are never called at run time can be filled in with arbitrary code, to make the static analysis of the application even more complex.

3 Class Hierarchy Flattening

This section presents the analyses and transformations needed to automatically transform the unprotected program into the one that is much harder to attack.

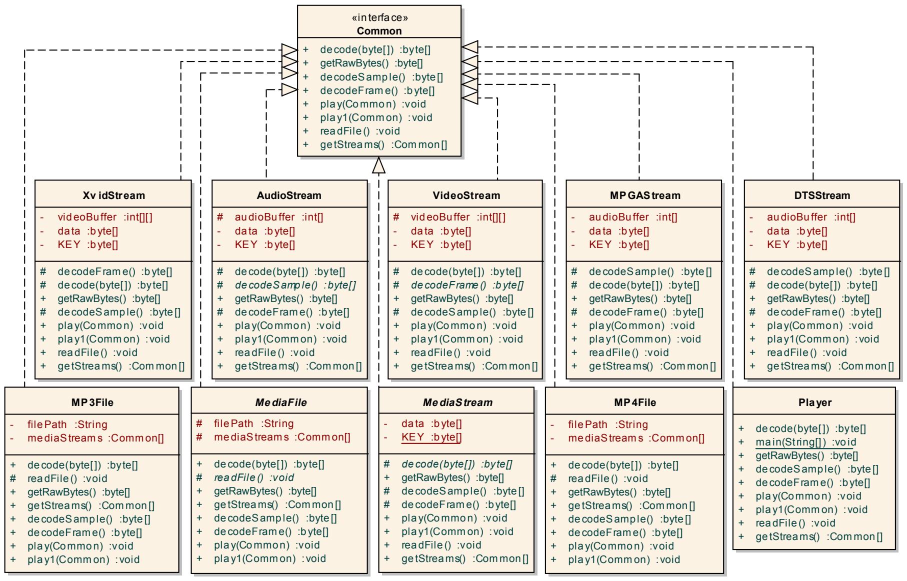


Fig. 3: Flattened class hierarchy of the media player

3.1 Basic Algorithm

The basic class hierarchy flattening (CHF) algorithm consists of five steps.

Step 1: Subtree selection We assume that each application consists of a set of *application classes* $\mathbb{A}$ that use or extend classes from a self-contained set of *library classes* $\mathbb{L}$ that includes `java.lang.Object`. Classes in $\mathbb{L}$ are never considered for transformation. $\mathbb{L}$ will usually correspond to the standard library, while $\mathbb{A}$ will contain all classes that make up the actual application. In this paper, $\text{sub}(x)$ denotes all subclasses of class x, and $\text{super}(x)$ its superclasses.

There is a subset $\mathbb{X} \subset \mathbb{A}$ of classes on which our CHF transformation is not applicable because changing those classes' position in the hierarchy can alter the program behavior. This includes classes on which reflective operations are performed such as `getInterfaces()` (which makes the program potentially dependent on the number of interfaces implemented by a class), `getSuperclass()`, `isAssignableFrom()`, `getMethod()`, etc.

As library classes cannot be rewritten, we cannot change their position in the hierarchy, nor can we adapt their methods' signatures, which typically involve library types themselves. To maintain type correctness, this implies that in general any application class $a \in \text{sub}(l)$ with $l \in \mathbb{L}$, needs to stay a subclass thereof. This further implies that we cannot make all application classes direct subclasses of `java.lang.Object`. This limitation is similar to the limitations imposed to several code refactoring transformations. Those limitations have been formalized in literature [31], so we will not repeat them here. As the classes in $\mathbb{X}$ cannot be moved in the hierarchy either, similar limitations apply to them.

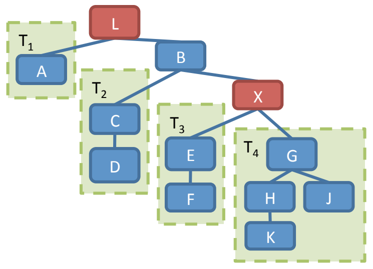


Fig. 5: Selected subtrees

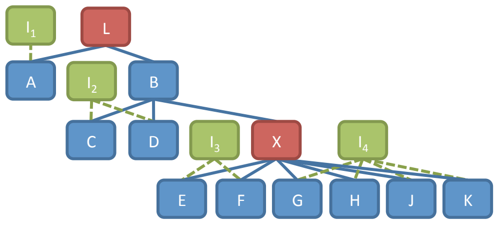


Fig. 6: Flattened class hierarchy subtrees

$$\mathbb{T} = \bigcup_{i=1..m} T_i$$

$$\forall i, j. T_i \cap T_j = \emptyset$$

$$T_i \subset \mathbb{A} \setminus \mathbb{X}$$

$$\forall c \in T_i.\text{sub}(c) \subset T_i$$

Fig. 4: Selection rules

For our purpose, we partition $\mathbb{A}$ into the set $\mathbb{T}$ of transformable classes and the set $\mathbb{N}$ of non-transformable classes. $\mathbb{T}$ is further partitioned into disjoint subtrees T_i according to the rules of Figure 4. They mainly express that each subtree T_i consists of a unique set of transformable classes for which the property holds that if T_i includes a class c, it also includes all of its subclasses. An example selection of subtrees is depicted in Figure 5. In the media player, the three subtrees correspond to the three subtrees of `java.lang.Object`.

Each subtree will then be transformed into one flat set of classes that all implement a common interface and that are all direct subclasses of the direct superclass of the tree's root. For the class hierarchy of Figure 5, the result with four new interfaces can be seen in Figure 6.

Step 2: Interface insertion To reach that final result, we first add the common supertype interfaces in two steps. For each subtree we encapsulate all instance fields declared in all classes of the subtree with getter and setter methods and rewrite public accesses to those fields into invocations of these getters and setters. This is done to provide access to instance fields declared in the subtree classes, even though interfaces cannot declare instance fields. Next, we create a new supertype interface for each subtree. This interface declares all instance methods of all classes in the subtree and is implemented by all classes of the subtree. Whenever an original class does not implement all the required methods of the interface, temporary dummy methods are added, some of which implement supercalls not to change the behavior of overriding methods.

Step 3: Subtree type abstraction Next, we rewrite the program such that it becomes independent of the subclass relations that will be removed from the class hierarchy. We replace all references to types in $\mathbb{T}$ by their corresponding interface supertype. In practice, this comes down to replacing the types of local variables, fields, array creations, and the types used in method signatures. The only time we still refer to the actual classes in the subtree is for object creation. An example of such a conversion of declarations can be seen in Figure 2, where various declarations have been replaced by the supertype `Common`.

As for cast operations, most of them are not needed anymore for static type correctness because interfaces instead of concrete types are used in declarations wherever possible. Moreover, we want to omit the remaining ones from the code not to reveal type information. So we replace them by code that tests a type and throws a `ClassCastException` whenever a run-time cast would have failed in the original program. To minimize the number of types that needs to be tested (and hence revealed in the code), we perform a points-to analysis on the original program [15, 23]. As such, our treatment of casts is similar to that in other code refactoring techniques that change type hierarchies [31].

Step 4: Subtree flattening Finally, we remove the inheritance relations between the classes in the subtrees, making subclasses independent of their superclasses. We traverse each subtree T_i in a breadth-first fashion. For each class $\texttt{c} \in T_i$, we execute the following steps for each direct subclass `d` of `c`:

1. copy the instance fields and concrete instance methods from `c` to `d`, renaming them if necessary to avoid collisions with original fields and methods of `d`;
2. rewrite the code in `d` such that it makes use of its own private copies of the methods and fields defined in `c`;
3. make `d` implement the same interfaces as `c`, to preserve assign compatibility between variables and fields of the interface types and objects of type `d`;
4. make `d` a sibling of `c` by setting its superclass to the superclass of `c`.

During this flattening, we replace many of the temporary dummy methods that were added when the interfaces got inserted. Not all of those methods are replaced, however. Consider, e.g., the `MediaStream` subtree. The interface for this subtree declares both the methods `decodeFrame()` and `decodeSample()`, and hence all classes originating from this subtree should implement those methods. That is why we inserted dummy implementations where necessary. In this case, some of the dummy implementations of `decodeFrame()` and `decodeSample()` in `DTSStream`, `MPGAStream`, and `XvidStream` are overwritten, but those in, e.g., `MediaStream`, `AudioStream`, and `VideoStream` are not. This poses no problem: As the non-overwritten methods were not present in the original program, and as we are not changing the behavior of the program, they will never be executed. We can therefore provide a dummy implementation for them, using nonsensical code [2] or carefully chosen code, as we propose in Section 3.2.

Step 5: instanceof The behavior of run-time type checks, introduced either explicitly by the programmer or automatically while handling casts during subtree type abstraction, depends on the specific organization of classes in the hierarchy. Before flattening the subtrees of Figure 1, `ms instanceof AudioStream` evaluated to `true` for `ms` pointing to objects of either type `DTSStream` or `MPGAStream`. In the flattened subtree, however, it evaluates to `false` for objects of those types.

To maintain the original behavior, we replace all occurrences of `instanceof` by a lookup in a table that encodes part of the original subtype relations, namely the part that is necessary to maintain the behavior with respect to `instanceof`. Each row in the lookup table initially corresponds to one of the instanceof expression $\texttt{o}_i$ `instanceof` $\texttt{A}_j$ in the program, while the columns correspond to (a superset of) the classes in the points-to set of all o_i. For the example program introduced in Section 2, the initial lookup table is given in Table 1.

	`XvidStream`	`AudioStream`	`VideoStream`	`MPGAStream`	`DTSStream`	`MP3File`	`MediaFile`	`MediaStream`	`MP4File`	`Player`
`ms instanceof AudioStream`	false	DC	DC	true	true	DC	DC	DC	DC	DC
`ms instanceof VideoStream`	true	DC	DC	false	false	DC	DC	DC	DC	DC
`mf instanceof MediaFile`	DC	DC	DC	DC	DC	true	DC	DC	true	false

Table 1: `instancoef` lookup table

As most of the classes will not occur in all points-to sets of all occurrences of `instanceof`, a considerable number of elements in the table will be "don't care" (DC) values. As is done for the optimization of multi-output boolean functions for optimizing integrated circuits [21], we can freely choose how to instantiate those DCs, i.e., replace them by `true` or `false`. For example, as `MPGAStream` and `DTSStream` have identical behavior, they likely originate from the same subtree. We can hide this by instantiating their DC values in a way that makes the classes' behavior look different, thus hiding an existing relation between two classes in the program. Alternatively, we can make `XvidStream` and `Player`, which are not related at all, look related by instantiating their DC values such that their behavior becomes identical. Likewise, we can replace the last two, different occurrences of `instanceof` by two identical ones by instantiating their DCs appropriately. This way, completely unrelated cast operations look as if they cast related types.

In short, by instantiating the DC values in the table, we can reduce its size and make unrelated classes and casts look related and vice versa. Furthermore, we can use hashing and white-box crypto techniques [6] to prevent static analysis of the table and involved code. Exploiting these opportunities is future work.

Once the final lookup table is constructed, each expression $\mathtt{o}_i$ `instanceof` $\mathtt{A}_j$ is replaced by a call `myChecker.isInstance(`$\mathtt{r}_{i,j}$`,`$\mathtt{o}_i$`.getClass())` where $\mathtt{r}_{i,j}$ is the row index of the lookup table entry that corresponds to the given instanceof expression.

3.2 Extensions

Several extensions to CHF can be considered.

Dummy methods - introducing differences/similarities During the subtree flattening, methods and fields are copied from parent classes into their children. This creates an opportunity for attackers to infer the original class hierarchy by means of diffing tools like Stigmata (http://stigmata.sourceforge.jp/). To distract such tools, we can introduce artificial differences or similarities by choosing appropriate dummy method bodies. By copying function bodies from unrelated classes, we can make unrelated classes look related and vice versa.

Interface merging CHF as described above binds each subtree to one interface. This gives attackers the possibility to infer information about the original class hierarchy from the use of interfaces. To limit the amount of information that can be inferred, we can merge multiple (unrelated) interfaces into a single one. Such merging can result in more dummy methods in the classes, however, and hence in considerable overhead. This can be observed in the flattened media player hierarchy in Figure 3, in which the three interfaces are already merged.

It is important to note that in general, the merging of interfaces needs to be limited to subtrees originating from within the same jar files. The merged interface can then be packaged in that same jar, such that custom class loaders, of which it is not known which jar files they can access in the original program, can find them precisely when and where they need them.

Object allocation obfuscation Even after interface merging, some statements expose detailed type information. After the allocation on line 26 in Figure 2, `as` points to an object of type `MPGAStream`. From this information, points-to analysis deduce points-to sets of many local variables. In turn, other analyses like call graph construction and program slicing will also regain some precision to the advantage of attackers. To prevent the propagation of precise type information at allocation sites, we can replace individual allocations by multiple ones by means of opaque predicates [24]. For example, line 26 can be replaced by

```
Common as;
if(condition1) as = new XvidStream(...);
else if(condition2) as = new DTSStream(...);
else if(condition3) as = new MPGAStream(audioData);
else as = new AVC1Stream(...);
```

in which the first two conditions opaquely evaluate to false, and the third one opaquely evaluates to true. Switch statements can also be used of course. Alternatively, we can introduce factories [12] that return all types that implement an interface. Factories are more stealthy [7] as they look more like regular, well-engineered code. Moreover, whereas context-insensitive coverage analysis suffices to detect that potential opaque predicates or switches only evaluate to one value, context-sensitive ones will be needed to obtain equally useful information from factories implemented in separate methods.

The effect of such object allocation obfuscations, when not undone by an attacker, will be that no points-to analysis, however complicated, will compute more precise results than the analysis based on class hierarchy analysis [10].

Combining flattening with other obfuscations CHF can be combined with several existing design obfuscations. CHF enables, e.g., more efficient class coalescing. Coalescing `MP3File` and `VideoStream` in Figure 1 with the algorithm proposed by Sosonkin et al. [28] would require `MediaFile` and `MediaStream` to be coalesced as well. This would increase the number of fields in all classes that inherit from the coalesced class by a factor two. After CHF, `MP3File` and

`VideoStream` can be coalesced without affecting the number of fields, and consequently the size of objects, of other classes.

CHF can also prepare a program for false factoring [8]. In Figure 3 all classes inherit directly from `java.lang.Object` and dependencies on the original inheritance relations have already been removed, so the classes can easily be reorganized in a fake hierarchy by inserting random superclasses.

4 Evaluation

We implemented CHF in Soot 2.5.0 [18, 32], an analysis and transformation framework for Java bytecode. As our tool rewrites the application bytecode that the developer has packaged in a collection of jar files, it does not require any changes to the application's source code.

Our implementation consists of two parts; a CHFTransformer and a refactoring toolkit. The CHFTransformer implements CHF as a Soot SceneTransformer, such that it can be applied together with Soot's other whole program transformations. Our refactoring toolkit provides a series of refactoring transformations, including *encapsulate field*, *rename field/method*, and variations of *push down field/method* and *extract interface* that were required to implement CHF [11].

To detect the set of non-transformable classes and to ensure that all Java features like reflection and dynamic class loading are handled correctly, we rely on TamiFlex, a tool developed specifically for facilitating the static analysis of Java programs that use such features [4]. As a profile-based tool, TamiFlex relies on the developer to provide program inputs that generate enough coverage. Alternatively, the developer can manually complement the coverage of TamiFlex with his knowledge of how the program depends on reflection and class loaders.

4.1 Benchmarks

We use the DaCapo 9.12 benchmark suite [3] to evaluate the protection-wise effectiveness and the performance-wise efficiency of CHF. This suite consists of 14 medium to large sized realistic applications. Because of space concerns, we report results for the four applications listed in Table 2. We opted for the "9.12 bach" release of the DaCapo suite because TamiFlex is particularly well tested on this version (http://dacapobench.org/soot.html). As can be seen in the table, for three out of four benchmarks the large majority of all classes is transformable. For eclipse, the number of transformable classes is much lower, because of restrictions imposed by dynamically generated classes.

For all benchmarks, we generated and evaluated $1 + 1 + 5 \times 10$ versions. The first, base version is the original bytecode that comes with the DaCapo suite, but with identifier names obfuscated [7]. This type of obfuscation is orthogonal to CHF; any Java obfuscator would apply it. We applied it for our evaluation baseline to obtain results for realistically obfuscated programs and to be able to present realistic overheads in term of code size and memory footprint, both of which heavily depend on the length of identifiers.

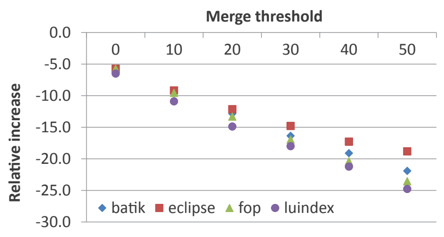
Benchmark	Description	# appl. types	# transf. classes	# jar files	code size (MB)	
					before IO	after IO
batik	Scalable Vector Graphics processor	4573	3505 (77%)	6	12.5	9.3
eclipse	non-GUI version of Eclipse IDE	5947	2258 (38%)	48	25.7	22.7
fop	XSL-FI to PDF conversion	4479	3349 (75%)	7	11.0	8.8
luindex	document indexing based on Lucene	633	526 (83%)	3	1.9	1.2

Table 2: Overview of DaCapo 9.12-bach benchmarks before and after Identifier Obfuscation (IO).

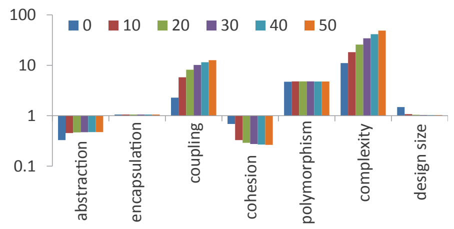
The second version was generated from the first one by our prototype implementation of the basic CHF algorithm as discussed in Section 3.1. Next, we extended the basic algorithm with interface merging (Section 3.2) and we generated program versions at different levels of interface merging. Given a merge threshold value $t \in \{10, 20, 30, 40, 50\}$, the extended tool iteratively and randomly picks interfaces in the program and merges them until all merged interfaces are implemented by at least t classes or until it can no longer find interfaces to merge within a jar. The latter occurs a lot for `eclipse`, of which the classes are partitioned over many more jar files. For each merge threshold value, the tool generates ten different program versions with ten different random seeds. When we present results for a level of merging in later charts, we always present the average result obtained for the ten versions at that level. In the charts, a merging threshold of 0 refers to the basic CHF algorithm without interface merging. In our proof-of-concept tool, the dummy method bodies are empty. Other extensions as described in Section 3.2 are left for future work. All generated program versions were type verified and proved to work correctly on the DaCapo inputs.

4.2 Results

Protection against Human Program Understanding As all obfuscation researchers, we face the problem of measuring the potency of our technique. And as in almost all of the literature (see, e.g., the literature discussed in Section 5), we know of no suitable metrics that directly measure the resistance to, e.g., reverse-engineering attacks. Therefore we instead rely on established software complexity metrics from the domain of software-engineering. In particular, we use the static QMOOD metrics from Bansiya et al. [1]. QMOOD stands for Quality Model for Object-Oriented Design. It includes a metric for understandability that is defined as a linear combination of other complexity metrics that measure different aspects of a design, including abstraction, encapsulation, coupling, cohesion, polymorphism, complexity and size [1]. This understandability metric is a relative metric that can only be used to compare two program versions. Given an original program with a normalized understandability score of -0.99 (as defined in [1]), less understandable versions will have lower scores. Figure 7(a) displays the relative understandability for the four benchmark programs. CHF clearly reduces human understandability significantly, with understandability dropping as more interfaces are merged. For `eclipse`, less interfaces got merged at higher merge thresholds because its classes are partitioned over more jar files. This results in a higher understandability than the other benchmarks.



(a) understandability results



(b) QMOOD breakdown for `batik`

Fig. 7: QMOOD understandability

Figure 7(b) shows the breakdown of `batik`'s understandability over its components for the six threshold levels. For other benchmarks, similar results are obtained. Each bar depicts the relative value of one metric compared to the value of that metric of the baseline program. It can be observed that CHF influences abstraction, coupling, cohesion, and complexity of the program, of its classes and of its class hierarchy. As such, the impact of CHF on the effort needed by an attacker to reverse-engineer and understand the program is multidimensional.

For all benchmarks, the variation in understandability score among the 10 program versions generated for each level of interface merging was at most 12%, the large majority of which was below 9%. This shows that interfaces can be merged in less or more confusing combinations, but also that the decrease in understandability is determined more by the level of merging than by the particular combinations merged.

Protection against Static Analysis Tools We measure the ability to confuse static analyses in terms of increases in points-to set sizes. In practice, the precision of many important client analyses, including call graph construction [14] and virtual call resolution, can drop significantly as the result of an imprecise points-to analysis.

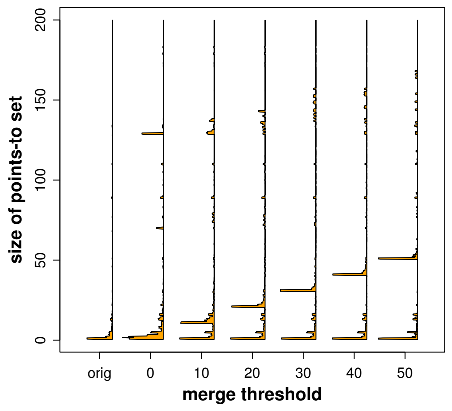


Fig. 8: Points-to set sizes in `batik`

At the same time, the analysis costs such as memory footprint and execution time increase with less precise points-to analyses because the constructed call graphs becomes bigger. Hence, reducing the precision of points-to analyses by causing them to return larger points-to sets, will directly reduce both the effectiveness and the efficiency of several static analyses that are fundamental for static code attacks. We made Soot compute the points-to sets with class hierarchy analysis [10], the points-to analysis that cannot be hampered by

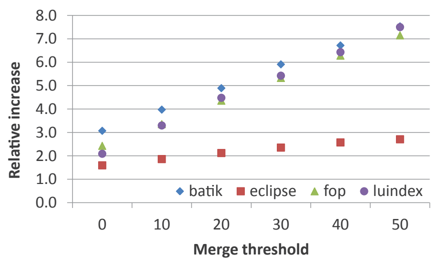


Fig. 9: Code size overhead

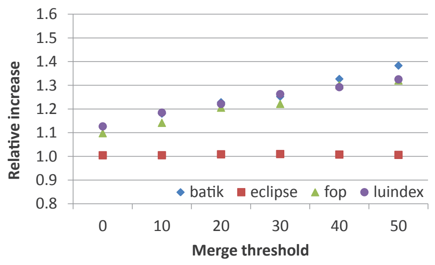


Fig. 10: Memory footprint overhead

object allocation obfuscation as discussed in Section 3.2. The histogram in Figure 8 depicts the distributions of points-to set sizes of all local variables and parameters in the methods of classes declared as (transformable or not) application types in the `batik` benchmark. For the other benchmarks, we observed very similar trends.

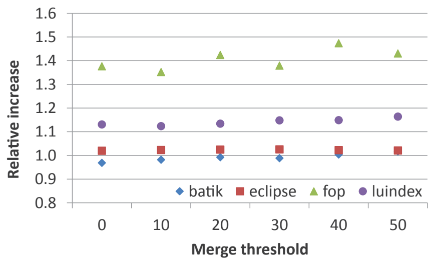
Clearly CHF increases the sizes of many points-to sets. In particular those points-to sets of variables declared as merged interface types grow with the number of classes implementing those interfaces. In all benchmarks, a considerable number of points-to sets does not grow even when more interfaces are merged. This follows from the fact that the increases are limited to the points-to sets of variables whose type is changed during CHF.

Not visible in the histograms, but similarly to what we observed for QMOOD, the points-to set size increases depend much more on merging threshold than on the particular combinations of interfaces merged.

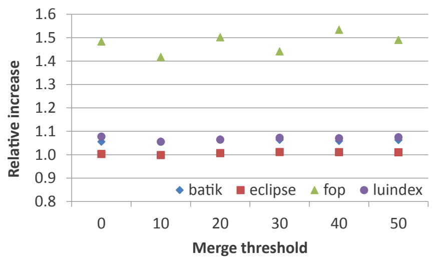
Overhead Figure 9 depicts the relative code size increase as a result of the basic flattening and interface merging. Overall, more interface merging implies more code. The increase, which results mainly from methods being duplicated and (mostly) empty dummy methods being added, varies from one benchmark to the other. The lower increase for `eclipse` is caused by its classes being partitioned over more jar files, as a result of which fewer interfaces got merged.

Figure 10 depicts the relative memory footprint increase observed with the Java SE Runtime Environment (build 1.6.0_30-b12) and the Java HotSpot 64-bit Server VM (build 20.6-b03) on standard runs consisting of 10 consecutive program executions in a benchmark harness on the default inputs. In general, the memory footprint overhead is a linear function of the code size overhead because classes and code are also stored in memory. The memory footprint overhead is an order of magnitude smaller, however, because there are many class instances (i.e, objects) allocated on the heap whose size is unaffected by CHF.

Finally, Figure 11 depicts the performance overhead in terms of the relative execution time increase. The overheads reported in Figure 11(a) include all 10 runs of the benchmarks in their harness. This includes the warm-up runs during which the JIT compiler is very active. As the code size grows with interface merging, so does the time spent by the JIT compiler. Figure 11(b) shows that during steady-state (i.e., when only the last of the 10 runs is considered during



(a) warm-up included



(b) steady-state

Fig. 11: Performance overhead

which almost no JIT compilation takes place anymore) the performance overhead no longer depends on the merging threshold value. For most benchmarks, the overhead is limited to less than 10%. For `eclipse`, the overhead is insignificant because most of its execution time is spent in non-transformable classes. The small variations in the observed execution time of `eclipse` are clearly within the expected noise range [13]. For `fop`, considerably more overhead remains. A performance analysis revealed that the getters and setters introduced during the interface insertion are the culprits.

5 Related work

To obfuscate the overall application design, and in particular its class hierarchy and the type information contained in the code, Sosonkin et al. proposed class coalescing, class splitting, and type hiding by introducing interface types and by replacing declarations of class types with declarations of those interfaces [28]. In its most extreme form, their class coalescing transformation can coalesce all transformable classes in the program into a single class, effectively removing the whole program design; beyond what CHF can achieve. For example, when all classes are coalesced, all points-to sets becomes singletons that contain all types in the program. In other words, points-to sets become completely useless. The main disadvantage of class coalescing is that the number of member fields in coalesced classes grows far beyond the number of original member fields in the original classes and all their superclasses. As a result, their instances also grow bigger, which will lead to a much larger memory footprint. The authors acknowledge this potential issue, but their experimental evaluation is limited to execution time measurements of relatively small programs (up to 307 classes). For those, they measure slow-downs up to 130% even with limited coalescing. Furthermore, their evaluation does not contain any criteria related to software protection, software understandability, or software complexity, and when limitations to the application of their transformations are observed, they hide behind tool maturity instead of investigating more fundamental issues. By contrast, this paper proposed an obfuscation that from the very start maximally removes the class hierarchy, and of which the code size, memory footprint, and (smaller) performance overhead are evaluated in detail, as well as its impact on program understandability, for a set of large, real-life programs (up to 5947 classes). Furthermore, rather than being immature, our prototype tool pushes the application of our obfuscation to the fundamental limits relating to external libraries, dynamic class loading and reflection.

The false factoring transformation proposed by Collberg et al. [8] refactors a program in such a way that two or more unrelated classes come to share a superclass, thereby giving the impression that they are related. We know of no public tool implementing this proposal or of any experimental evaluation of it.

Given a set of transformable classes, the obfuscation techniques introduced by Sakabe et al. [24] first changes the signature of all methods in the classes such that each class implements the same set of overloaded methods. These methods are then defined in an interface implemented by the classes and used in declarations instead of the original classes. To hide the actual type of objects bound to variables of the interface type, they propose to replace single object creations by a set of object creations guarded by opaque predicates. CHF as presented here is to a certain degree complementary, as explained in Section 3.2.

In the field of software refactoring, Snelting and Tip [26, 27] presented a method for analyzing and reengineering class hierarchies by extracting information on the use of an application's class hierarchy, from which they construct a concept lattice that provides insights on how to improve the hierarchy to better match the way the classes interact. Their analysis can detect where class members can be moved to a subclass or identify where it is beneficial to split classes. This analysis has been extended and implemented in the refactoring tool KABA [25, 29, 30]. This tool uses the results from the concept analysis to present several refactorings to the user, who can then interactively modify the class hierarchy. Potentially, Snelting and Tip's work could help an attacker find related classes in a flattened hierarchy by allowing him to see through the smokescreen of specially crafted dummy method implementations and by detecting unrelated classes implementing merged interfaces. It remains an open question to assess to which extent their tool would be useful in practice.

6 Conclusions and Future Work

This paper presented class hierarchy flattening, an obfuscating program transformation for object-oriented programs written in managed code languages. The transformation removes the class hierarchy to the extent possible to hide the overall application design. We presented the basic technique and possible extensions. Together with the basic algorithm, one of those extensions, called interface merging, was evaluated extensively on large real-world programs. While several aspects of the experimental results deserve further analysis, they clearly demonstrate that class hierarchy flattening provides measurable protection against at least some forms of human understandability and automated program analysis. This protection is achieved at relatively low levels of run-time overhead.

Our future work will concentrate on more focused interface merging strategies to outperform random merging, on extending the basic protection as discussed in the paper, and on a more extensive evaluation involving more security metrics, including diffing-based metrics, and more complex points-to analyses.

References

1. Bansiya, J., Davis, C.G.: A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Eng. **28**(1) (January 2002) 4–17
2. Batchelder, M., Hendren, L.J.: Obfuscating Java: The most pain for the least gain. In: Proc. CC. (2007) 96–110
3. Blackburn, S.M., McKinley, K.S., et al: Wake up and smell the coffee: evaluation methodology for the 21st century. Commun. ACM **51**(8) (August 2008) 83–89
4. Bodden, E., Sewe, A., et al: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: Proc. ICSE. (2011) 241–250
5. Chan, J.T., Yang, W.: Advanced obfuscation techniques for Java bytecode. Journal of Systems and Software **71**(1-2) (2004) 1–10
6. Chow, S., Eisen, P.A., Johnson, H., van Oorschot, P.C.: White-box cryptography and an AES implementation. In: Proc. SAC. (2003) 250–270
7. Collberg, C., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley Professional (2009)
8. Collberg, C., Thomborson, C., Douglas, L.: A taxonomy of obfuscating transformations. Technical report, University of Auckland (1997)
9. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proc. ACM POP. (1998) 184–196
10. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Proc. ECOOP. (1995) 77–101
11. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
12. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
13. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous java performance evaluation. In: Proc. ACM OOPSLA. (2007) 57–76
14. Grove, D., Chambers, C.: A framework for call graph construction algorithms. ACM Trans. Program. Lang. Syst. **23**(6) (November 2001) 685–746
15. Hind, M., Pioli, A.: Evaluating the effectiveness of pointer alias analyses. Science of Comp. Programming **39**(1) (2001) 31–55
16. Holst, S.: Assessing and managing security risks unique to Java and .NET. ISSA Journal (2009)
17. Hou, T., Chen, H., Tsai, M.: Three control flow obfuscation methods for Java software. IEE Proceedings-Software **153**(2) (APR 2006) 80–86
18. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The Soot framework for Java program analysis: a retrospective. In: Proc. CETUS 2011. (October 2011)
19. Majumdar, A., Thomborson, C.D.: Manufacturing opaque predicates in distributed systems for code obfuscation. In: Proc. ACSC. (2006) 187–196
20. Majumdar, A., Thomborson, C.D., Drape, S.: A survey of control-flow obfuscations. In: ICISS. (2006) 353–356
21. McCluskey, E.: Introduction to the theory of switching circuits. McGraw Hill Text (1965)
22. Palsberg, J., Krishnaswamy, S., Kwon, M., Ma, D., Shao, Q., Zhang, Y.: Experience with software watermarking. In: Proc. ACSAC. (2000) 308–316

23. Ryder, B.G.: Dimensions of precision in reference analysis of object-oriented programming languages. In: Proc. CC 2003, Warsaw, Poland (2003) 126–137
24. Sakabe, Y., Soshi, M., Miyaji, A.: Java obfuscation approaches to construct tamper-resistant object-oriented programs. IPSJ Digital Courier **1** (2005) 349–361
25. Snelting, G., Streckenbach, M.: KABA: Automated refactoring for improved cohesion. In: Proc. of the first Workshop on Refactoring Tools. (2007) 1–2
26. Snelting, G., Tip, F.: Reengineering class hierarchies using concept analysis. In: Proc. ACM FSE. (1998) 99–110
27. Snelting, G., Tip, F.: Understanding class hierarchies using concept analysis. ACM Trans. Program. Lang. Syst. **22**(3) (May 2000) 540–582
28. Sosonkin, M., Naumovich, G., Memon, N.: Obfuscation of design intent in object-oriented applications. In: Proc. ACM DRM. (2003) 142–153
29. Streckenbach, M.: KABA - a system for refactoring Java programs. PhD thesis, Universität Passau (2005)
30. Streckenbach, M., Snelting, G.: Refactoring class hierarchies with KABA. In: Proc. ACM OOPLSA. (2004) 315–330
31. Tip, F., Furher, R., Kieżun, A., Ernst, M., Balaban, I., De Sutter, B.: Refactoring using type constraints. ACM Trans. Program. Lang. Syst. **33**(3) (2011) 9:1–9:47
32. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: Proc. CASCON. (1999) 125–135
33. Venkatraj, A.P.R.: Program obfuscation. Master's thesis, University of Arizona (2003)
34. Zhou, Y., Main, A., Gu, Y.X., Johnson, H.: Information hiding in software with mixed boolean-arithmetic transforms. In: Proc. WISA. (2007) 61–75