Pushing Java Type Obfuscation to the Limit

Christophe Foket, Bjorn De Sutter, *Member, IEEE Computer Society,*
and Koen De Bosschere, *Member, IEEE Computer Society.*

***Abstract*—Bytecoded .Net and Java programs reveal type information through encoded type hierarchies, casts, field declarations and method signatures. This facilitates bytecode verification, but it also helps reverse engineers. To obfuscate the type information, we combine three transformations. *Class hierarchy flattening* removes as much of the type hierarchy from programs as possible. *Interface merging* and *object factory insertion* further remove type information from casts, method signatures, and object creation sites. We evaluate these techniques with a prototype tool for Java bytecode. On real-life programs from the DaCapo benchmark suite, we demonstrate that our approach effectively hinders human and tool analysis with limited overhead.**

***Index Terms*—Java bytecode, obfuscation, class hierarchy, type information, points-to sets, understandability, performance**

I. Introduction

Reverse engineering and modification of managed bytecode are well-understood and common practices, with many legitimate goals [1]. Malicious developers can abuse them, however, to attack Java and .NET applications with the goals of software piracy, software IP theft, data theft, and malware injection. Their attacks benefit from the fact that virtual machines require symbolic information to execute the bytecode and to manage the data. Type information constitutes an important part of this, as it enables just-in-time compilation, garbage collection, reflection and bytecode verification. At the same time, this type information is also what makes managed code easier to understand, reverse engineer, modify, reuse and steal.

Many obfuscation techniques have been proposed to harden reverse engineering attacks. Some prevent automatic decompilation [2], [3]. Some obfuscate data flow [4], [5] or control flow [3], [4], [6], [7], [8], [9], [10], [11]. Others simply omit human-readable, meaningful identifiers [2], [12]. Finally, a few have proposed obfuscating the overall application design by altering the application's type hierarchy [13]. The latter obfuscations aim for the opposite of classic code refactoring [14], [15].

We take design obfuscation one step further. Instead of merely modifying an application's type hierarchy, we propose class hierarchy flattening (CHF) to get rid of it altogether. CHF strives to maximally remove subtype relations, resulting in a hierarchy in which classes are siblings rather than subtypes and supertypes. To avoid that method signatures, casts, and object creation sites still yield type information in flattened code, we combine CHF with two additional transformations: interface merging (IM) and object factory insertion (OFI).

All authors are with the Computer Systems Lab, Department of Electronics and Information Systems, Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium. E-mail: {cfoket,bjorn.desutter,kdb}@elis.ugent.be.

The authors 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 for the STEVIN Supercomputer Infrastructure on which we carried out part of this work.

The major contributions of this paper are the following.

- We present CHF. In combination with the two other transformations it enables, CHF obfuscates much more type information than existing obfuscations.
- We present effective methods and heuristics to limit the overhead and to trade it for the level of protection.
- We present a tool flow for applying the transformations on complex, real-world applications.
- We are the first to evaluate and report obfuscations of this complexity on large, real-world applications.
- Our evaluation includes metrics related to human code understanding as well as to automated static analysis tools that help attackers reverse-engineer bytecode.

The remainder of this paper is structured as follows. Section II illustrates the obfuscations on an example program. Sections III, IV, and V discuss CHF, IM, and OFI respectively. We evaluate them in Section VI and address their correctness in Section VII. Related work is the topic of Section VIII. Section IX concludes and discusses future work.

II. Rationale: an Example Program

An example media player illustrates the issues we tackle. It consists of three parts: the player initializer, support for media files, and support for media streams in those files. Fig. 1 shows the corresponding class hierarchy subtrees. Fig. 2(a) illustrates their interaction. For the sake of clarity, we use meaningful method and type identifiers. In a real obfuscated program, they would of course be replaced by meaningless ones [2], [12].

The main method of class Player creates an array of MediaFile objects to be played (l. 10). It then queries them for their media streams (l. 12), which are initialized by accessing the file with the readFile method. Fig. 2(a) shows this for the MP3File class, which represents MP3 files containing MPEG audio streams. During playback, the player checks the run-time type of the MediaStream objects (ll. 13 & 15) to decide where they need to be output. They are either cast to AudioStream or VideoStream, such that the correct play method is invoked (ll. 14 & 16). The play methods essentially output the raw bytes of the media streams to a specific output device. Those bytes are obtained, decrypted (ll. 33–34) and decoded (l. 35) with the getRawBytes method declared in MediaStream. Because the decoding process is different for each type of stream, the decode method is declared as abstract, and is 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.

From a software-engineering perspective, the code is well structured. The inheritance relations are meaningful and code shared between different classes is located in a common

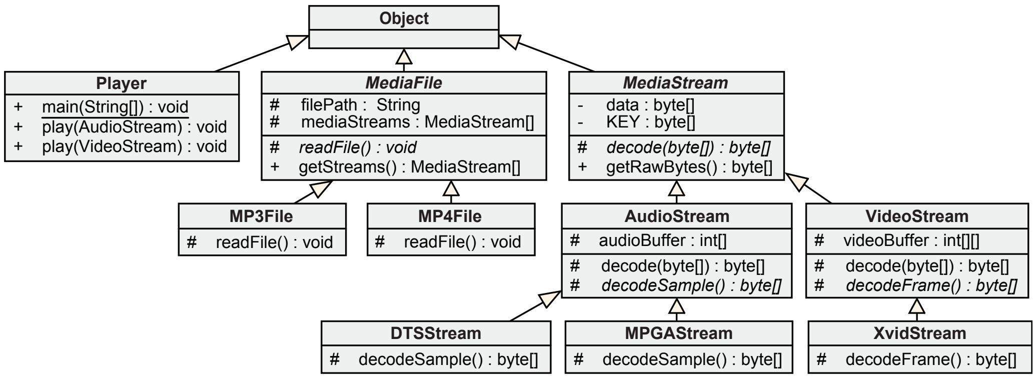

Fig. 1: Standard UML representation of the class hierarchy of a simple DRM media player.

```
 1 public class Player {
 2   public void play(AudioStream as) {
 3     /* send as.getRawBytes() to audio device */
 4   }
 5   public void play(VideoStream vs) {
 6     /* send vs.getRawBytes() to video device */
 7   }
 8   public static void main(String[] args) {
 9     Player player = new Player();
10     MediaFile[] mediaFiles = ...;
11     for (MediaFile mf : mediaFiles)
12       for (MediaStream ms : mf.getStreams())
13         if (ms instanceof AudioStream)
14           player.play((AudioStream)ms);
15         else if (ms instanceof VideoStream)
16           player.play((VideoStream)ms);
17   }
18 }
19 public class MP3File extends MediaFile {
20   protected void readFile() {
21     InputStream inputStream = ...;
22     byte[] data = new byte[...];
23     inputStream.read(data);
24     AudioStream as = new MPGAStream(data);
25     mediaStreams = new MediaStream[]{as};
26     return;
27   }
28 }
29 public abstract class MediaStream {
30   public static final byte[] KEY = ...;
31   public byte[] getRawBytes() {
32     byte[] decrypted = new byte[data.length];
33     for (int i = 0; i < data.length; i++)
34       decrypted[i] = data[i] ^ KEY[i];
35     return decode(decrypted);
36   }
37   protected abstract byte[] decode(byte[] data);
38 }
```

(a) original code

```
 1 public class Player implements Common {
 2   public void play(Common as) {
 3     /* send as.getRawBytes() to audio device */
 4   }
 5   public void play1(Common vs) {
 6     /* send vs.getRawBytes() to video device */
 7   }
 8   public static void main(String[] args) {
 9     Common player = new Player();
10     Common[] mediaFiles = ...;
11     for (Common mf : mediaFiles)
12       for (Common ms : mf.getStreams())
13         if (myCheck.isInst(0, ms.getClass()))
14           player.play(ms);
15         else if (myCheck.isInst(1, ms.getClass()))
16           player.play1(ms);
17   }
18 }
19 public class MP3File implements Common {
20   public void readFile() {
21     InputStream inputStream = ...;
22     byte[] data = new byte[...];
23     inputStream.read(data);
24     Common as = new MPGAStream(data)
25     mediaStreams = new Common[]{as};
26     return;
27   }
28 }
29 public class MediaStream implements Common {
30   public static final byte[] KEY = ...;
31   public byte[] getRawBytes() {
32     byte[] decrypted = new byte[data.length];
33     for (int i = 0; i < data.length; i++)
34       decrypted[i] = data[i] ^ KEY[i];
35     return decode(decrypted);
36   }
37   public byte[] decode(byte[] data){ ... }
38 }
```

(b) partially obfuscated type information

```
 1 public class Player implements Common {
 2   public byte[] merged1(Common as) {
 3     /* send as.getRawBytes() to audio device */
 4   }
 5   public Common[] merged2(Common vs) {
 6     /* send vs.getRawBytes() to video device */
 7   }
 8   public static void main(String[] args) {
 9     Common player = CommonFactory.create(...);
10     Common[] mediaFiles = ...;
11     for (Common mf : mediaFiles)
12       for (Common ms : mf.getStreams())
13         if (myCheck.isInst(0, ms.getClass()))
14           player.merged1(ms);
15         else if (myCheck.isInst(1, ms.getClass()))
16           player.merged2(ms);
17   }
18 }
19 public class MP3File implements Common {
20   public byte[] merged1() {
21     InputStream inputStream = ...;
22     byte[] data = new byte[...];
23     inputStream.read(data);
24     Common as = CommonFactory.create(...);
25     mediaStreams = new Common[]{as};
26     return data;
27   }
28 }
29 public class MediaStream implements Common {
30   public static final byte[] KEY = ...;
31   public byte[] getRawBytes() {
32     byte[] decrypted = new byte[data.length];
33     for (int i = 0; i < data.length; i++)
34       decrypted[i] = data[i] ^ KEY[i];
35     return decode(decrypted);
36   }
37   public byte[] decode(byte[] data){ ... }
38 }
```

(c) fully obfuscated type information

Fig. 2: Partial implementations of the Player, MediaStream and MP3File classes. Transformed code is underlined.

superclass. While we could have factored out casts and runtime type checks, we did not do so for didactic purposes.

From a security perspective, there are some issues. First, the hierarchy informs attackers about the abstraction levels of the classes' functionalities. Classes higher in the hierarchy typically provide more abstract functionality. Secondly, code reuse through inheritance enables attacks in which compromising one class can compromise all of its subclasses. All media streams are decrypted with MediaStream.getRawBytes(). When an attacker reverse-engineers this method, he can decrypt all supported media stream types. Finally, we observe that even though local variables are untyped in bytecode, the code still reveals type information through method signatures, casts, and object creation sites. For example, the allocation of a Player on l. 9 allows a type inference tool [16] to narrow the type of the player variable to Player. The instanceof checks and casts on ll. 13–16 also restrict the possible types to which the variable ms can point. This abundance of type information is important for an attacker because it simplifies his mental understanding and his tools' formal models of the code. In compiler terminology, it reduces points-to set sizes and it simplifies the call graph by omitting unrealizable edges [17].

These issues can be solved by rewriting the well-structured hierarchy into the unstructured collection of Fig. 3. To determine how classes are related, an attacker can then no longer rely on a hierarchy. He instead has to analyze all classes. Furthermore, as all classes are provided with a (diversified) copy of all fields and methods declared in their former superclasses, they have become independent. Code is no longer shared between related classes, so one can no longer attack many classes at once by patching their common superclass.

Code analysis has also become harder. Fig. 2(c) displays much less type information than the original code. All declarations declare type Common, all invoked methods are imple-

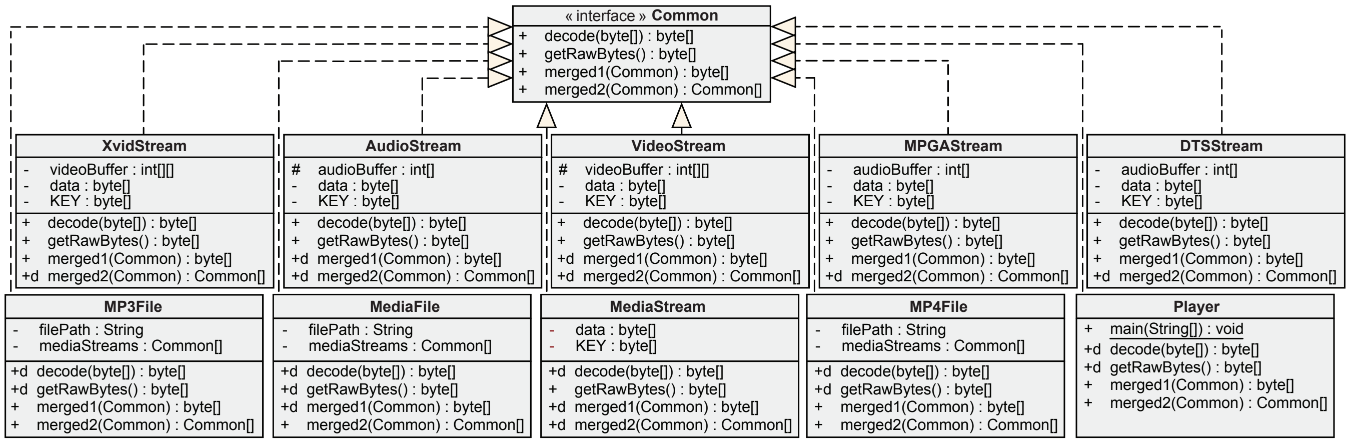
« interface » Common
+ decode(byte[]) : byte[]
+ getRawBytes() : byte[]
+ merged1(Common) : byte[]
+ merged2(Common) : Common[]
XvidStream
- videoBuffer : int[][]
- data : byte[]
- KEY : byte[]
+ decode(byte[]) : byte[]
+ getRawBytes() : byte[]
+ merged1(Common) : byte[]
+d merged2(Common) : Common[]
AudioStream
# audioBuffer : int[]
- data : byte[]
- KEY : byte[]
+ decode(byte[]) : byte[]
+ getRawBytes() : byte[]
+d merged1(Common) : byte[]
+d merged2(Common) : Common[]
VideoStream
# videoBuffer : int[][]
- data : byte[]
- KEY : byte[]
+ decode(byte[]) : byte[]
+ getRawBytes() : byte[]
+d merged1(Common) : byte[]
+d merged2(Common) : Common[]
MPGAStream
- audioBuffer : int[]
- data : byte[]
- KEY : byte[]
+ decode(byte[]) : byte[]
+ getRawBytes() : byte[]
+ merged1(Common) : byte[]
+d merged2(Common) : Common[]
DTSStream
- audioBuffer : int[]
- data : byte[]
- KEY : byte[]
+ decode(byte[]) : byte[]
+ getRawBytes() : byte[]
+ merged1(Common) : byte[]
+d merged2(Common) : Common[]
MP3File
- filePath : String
- mediaStreams : Common[]
+d decode(byte[]) : byte[]
+d getRawBytes() : byte[]
+ merged1(Common) : byte[]
+ merged2(Common) : Common[]
MediaFile
- filePath : String
- mediaStreams : Common[]
+d decode(byte[]) : byte[]
+d getRawBytes() : byte[]
+d merged1(Common) : byte[]
+ merged2(Common) : Common[]
MediaStream
- data : byte[]
- KEY : byte[]
+d decode(byte[]) : byte[]
+ getRawBytes() : byte[]
+d merged1(Common) : byte[]
+d merged2(Common) : Common[]
MP4File
- filePath : String
- mediaStreams : Common[]
+d decode(byte[]) : byte[]
+d getRawBytes() : byte[]
+ merged1(Common) : byte[]
+ merged2(Common) : Common[]
Player
+ main(String[]) : void
+d decode(byte[]) : byte[]
+d getRawBytes() : byte[]
+ merged1(Common) : byte[]
+ merged2(Common) : Common[]

Fig. 3: Obfuscated class hierarchy of the media player.

mented by all classes, and all casts are gone. An obfuscated, typeless isInstance method replaces instanceof, and factories returning Common replace type-specific allocations. These factories can be obfuscated internally, such that static analysis cannot determine the precise type of the returned objects. As a result, call graph construction [18], points-to analyses [17] and type inference [16] will yield less precise results.

Furthermore, as all classes now implement the whole Common interface, many of them now implement more methods. For example, in the obfuscated program all classes implement merged1, which replaces play, decodeSample, decodeFrame, and readFile. An attacker's static analysis cannot determine that of all ten implementations of merged1, only six will actually be executed. In AudioStream, VideoStream, MediaStream and MediaFile, the merged1 methods are dummies (marked with the non-standard d in Fig. 3) that can be filled with arbitrary code to complicate static analysis even further.

In the next sections, we discuss the stepwise code obfuscation. CHF (Sect. III) first replaces the type hierarchy by a flat collection of classes. In doing so, CHF introduces a single interface for each subtree of the original class hierarchy. Because of the different interfaces, the flattened program then still encodes a considerable amount of type information. With IM (Sect. IV), we merge separate interfaces into a common one. Method signatures then feature less diverse types and many casts can be removed. This leads to a reduction in type information and enables OFI (Sect. V) to further remove type information from object allocation sites for optimal protection.

III. Class Hierarchy Flattening

The goal of CHF is to remove as many subtype relations from a class hierarchy as possible. For example, in our media player MP3File and MP4File should no longer inherit from MediaFile. In practice, however, not all subtype relations can be removed. Any class hierarchy transformation is constrained by type correctness requirements imposed by external libraries that cannot be transformed, and by uses of reflection that cannot be easily transformed. CHF therefore proceeds in six steps. First, subtrees from a class hierarchy are selected that can be flattened. In five following steps, all necessary transformations are performed to flatten the selected subtrees.

A. Subtree selection

Assume that an 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. $\mathbb{L}$ usually corresponds to the standard library; $\mathbb{A}$ contains all classes that make up the actual application. Let $t_s, t^s : (\mathbb{A} \cup \mathbb{L}) \mapsto (\mathbb{A} \cup \mathbb{L})^*$ be the functions that map a class x onto the sets $t_s(x)$ and $t^s(x)$ of all x's (transitive) subclasses and superclasses, resp., x included.

As we cannot rewrite external library classes, we cannot change their position in the hierarchy, nor can we adapt their method signatures. To maintain type correctness, this implies that any application class a in $t_s(l)$ with $l \in \mathbb{L}$ needs to stay a subclass of l. This is similar to limitations imposed on other refactorings. Those limitations have been formalized in literature [15], so we do not repeat them here. Furthermore, CHF is not applicable to a subset of classes $\mathbb{X} \subset \mathbb{A}$ because changing those classes' position in the hierarchy could alter the program behavior. This includes classes on which the program might (depending on the input) perform reflective operations such as getInterfaces() (which can make the program dependent on the number of interfaces implemented by a class), getSuperclass(), isAssignableFrom(), getMethod(), etc. There can also be practical reasons for not flattening some classes. Our prototype tool presented in Sect. VI considers classes of which multiple different or identical definitions exist on the class path as non-transformable, as well as classes that implement java.lang.Throwable. The latter are usually exception classes. Implementing tool support for these types of classes would require a major engineering effort and not buy much in terms of obfuscation, so our tool simply adds them to $\mathbb{X}$ as non-transformable. All classes in $\mathbb{X}$ and their subclasses face similar limitations as those in $\mathbb{L}$ and their subclasses.

So we partition $\mathbb{A}$ into set $\mathbb{T}$ of transformable classes and set $\mathbb{X}$ of non-transformable classes. $\mathbb{T}$ is further partitioned into disjoint subtrees T_i according to the following four rules:

1) $\mathbb{T} = \bigcup_{i=1..m} T_i$ 2) $\forall i : T_i \subset \mathbb{A} \setminus \mathbb{X}$
3) $\forall i, j : \quad T_i \cap T_j = \emptyset$ 4) $\forall c \in T_i : t_s(c) \subset T_i$

They express that each subtree T_i consists of a unique set of transformable classes such that if T_i includes a class c, it also includes all of its subclasses. Fig. 4 depicts the selection of

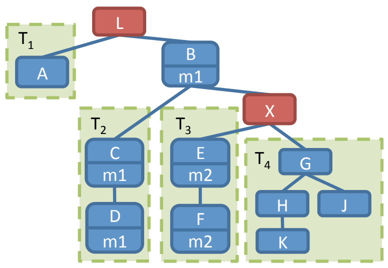

Fig. 4: Selected subtrees in a class hierarchy.

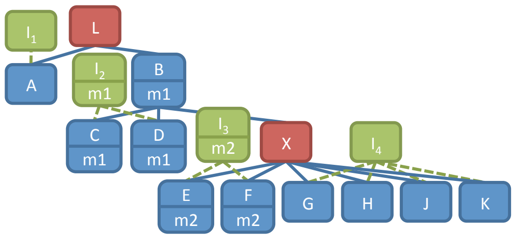

Fig. 5: Flattened class hierarchy subtrees.

four subtrees in a hierarchy with an external library class L and a non-transformable application class X. In the media player of Fig. 1, the three subtrees of java.lang.Object are selected.

In steps B–F each subtree will be further transformed into a flat set of classes that become direct subclasses of the direct superclass of the tree's root. For each flat set of classes, we insert an interface they implement. Fig. 5 shows the result with four new interfaces for the original class hierarchy of Fig. 4.

B. *Preparing subtrees for flattening*

We prepare subtrees in three steps. First, we encapsulate instance fields in subtree classes with getters and setters, and replace field accesses by calls to those getters and setters. This provides access to instance fields in the subtree classes, even though interfaces cannot declare instance fields. Secondly, we make each class functionally independent of its superclasses. We traverse each subtree T_i breadth-first. For each class $c \in T_i$ and each direct subclass d of c we first 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. For constructors, which cannot be renamed, we avoid collisions by adding artificial, distinguishing parameters. We then rewrite field references and super calls in d and d's subclasses such that they reference d's copies. References from outside d and its subclasses need not be rewritten. First, external field references have already been replaced by getters and setters. Secondly, when a method needs to be renamed or needs to get a distinguishing parameter to avoid a signature collision, this implies that shadowing already prevented external references to the original method in c.

C. *Interface insertion*

For each subtree we create a new supertype interface, and insert it into the directory or archive that contains the root class of the subtree. The interface declares all instance methods of all classes in the subtree and is implemented by all its classes. Whenever an original class does not implement all the required methods of the interface, dummy methods are added. Since these 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 nonsensical implementations for them so as to confuse static analyses.

For the media player, we create three interfaces: Common1, Common2, and Common3, corresponding to the subtrees rooted at Player, MediaFile, and MediaStream.

```
public class Player implements Common1 {
 public void play(Common3 as) { ... }
 public void play1(Common3 vs) { ...}
 public static void main(String[] args) {
   Common1 player = new Player();
   Common2[] mediaFiles = ...;
   for (Common2 mf : mediaFiles)
     for (Common3 ms : mf.getStreams())
       if (ms instanceof AudioStream)
         player.play(ms);
       else if (ms instanceof VideoStream)
         player.play1(ms);
 }
}
```

Fig. 6: Intermediate obfuscated Player class.

D. *Subtree type abstraction*

Next, we make the program independent of the subclass relations that we will remove in the next step. We replace all references to types in $\mathbb{T}$ by their corresponding interface supertypes. 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 and dynamic type checks. Fig. 6 shows a partially obfuscated version of the Player class. Various declarations have been replaced by the three supertypes Common1, Common2, and Common3, but new and instanceof expressions have not yet been replaced.

This abstraction operation sometimes requires method renaming. The two play methods in Fig. 2(a) had different parameter types, but they have identical types in Fig. 6. So one of them is renamed into play1 to differentiate them.

As for cast operations, many of them can be omitted because they have become superfluous after interfaces replaced concrete types in declarations. Not to reveal any type information, we also want to omit the remaining ones. So we replace them by code that tests a type with instanceof and throws a ClassCastException whenever a run-time cast would have failed in the original program. To minimize the number of types that need to be tested and hence revealed in the code, we perform a points-to analysis on the original program [19], [20]. This treatment of casts is similar to that in other code refactoring techniques that change type hierarchies [15].

We should also note that at some program points, the transformation requires us to add casts. This happens whenever an object is passed to a library function as a parameter. In Fig. 4, consider a method void m0(L x) of library class L. In the original program, the following sequence is valid: L o = ...; A a = ...; o.m0(a); After rewriting the declarations, we have to insert a cast: L o = ...; I1 a = ...; o.m0((L)a); This cast is needed for type correctness, but does not provide attackers additional type information, as the type in the cast was already present in the library method's signature anyway.

Such casts cannot be used for arrays, however. For the same hierarchy and a library method sort(L[] x), we cannot rewrite C[] c = new C[1]; c[0] = new D(); sort(c); into I2[] c = new I2[1]; c[0] = new D(); sort((L[])c); because the dynamic cast in the rewritten code will throw a ClassCastException. Likewise, we cannot rewrite G[] g = new H[1]; g[0] = new J(); into I4[] g = new I4[1]; g[0] = new J(); The former throws an ArrayStoreException; the latter does not.

Point-to set information for call sites of external methods and for array store operations allow us to identify the program points where these exceptions will never be thrown in the original and in the rewritten program. At the remaining program points, the exceptions may potentially be thrown in the original but not in the rewritten program or vice versa if the type relation between the involved arrays and objects changes because of flattening. To prevent this from happening, it suffices to treat a class as non-transformable if an array of that class occurs in a points-to set at such a program point.

E. Subtree flattening

We can now remove the class inheritance relations within the subtrees. Traversing each subtree T_i breadth-first, for each class $c \in T_i$ and each direct subclass d of c we first 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. Next, we make them siblings by setting d's superclass to c's.

Except for dynamic type checks, the program now no longer depends on the subtype relations between classes.

F. Converting instanceof

The behavior of run-time type checks inserted by the programmer as casts or in step D, depends on the structure of the class hierarchy. Before flattening the subtrees of Fig. 1, the expression ms instanceof AudioStream on line 13 of Fig. 2(a) 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 preserve the program semantics, we replace all occurrences of instanceof by table lookups. The table encodes at least those original subtype relations necessary to preserve the behavior of the instanceof occurrences. Each row in the table initially corresponds to one of the expressions o_i instanceof A_j in the original program. The columns correspond to the classes in the points-to sets computed for all o_i. For the media player, Table I represents the initial table. In real programs, the table will be much bigger. To mitigate analysis by attackers, the table can be inflated by adding additional classes as columns and by adding rows for dummy instanceof operations injected into the dummy methods inserted in step B.

As most of the classes will typically not occur in all points-to sets of all instanceof occurrences, a considerable number of elements in the table will be "don't care" (DC) values. As is done in the optimization of multi-output boolean functions for optimizing circuits [21], we can freely choose between true or false, to replace each DC. In our case, we want to minimize the amount of useful type information in the table. Consider, for example, the MPGAStream and DTSStream classes. The similarity between their columns in Table I indicates that they originate from the same subtree. To hide this from attackers analyzing an inflated table, we can instantiate their DC values in a way that makes the classes' cast behavior look different. Alternatively, we can make, e.g., XvidStream and Player, which are not related at all, look related by choosing their DC values such that their behavior becomes identical. Likewise, we can merge the last two, different occurrences of instanceof by choosing their DCs appropriately. This way, originally completely unrelated cast operations look as if they cast related types. In short, by choosing the DC values in the possibly inflated table, we can again reduce its size and make unrelated classes and casts look related and vice versa. Furthermore, we can use hashing and white-box crypto techniques [22] to prevent static analysis of the table and involved code.

Each expression o_i instanceof A_j is then replaced by a call myCheck.isInst($\mathsf{r}_{i,j}$,o_i.getClass()) where $\mathsf{r}_{i,j}$ is the (possibly encrypted or hashed) row index of the lookup table entry that corresponds to the given instanceof expression. Lines 13 and 15 of Fig. 2(b) show the results for the media player.

IV. Interface Merging

In the code in Fig. 6 much type information can still be inferred from declarations and method signatures. This will lead to small points-to sets and more precise analyses. Additionally, an attacker can determine which classes belonged to the same subtrees in the original program based on the subtype relations between classes and interfaces.

To limit the amount of available type information, we can merge multiple unrelated interfaces into a single one. For the media player example, the interfaces Common1, Common2 and Common3 can be merged into an interface Common that declares all their methods. The result is shown in Fig. 7 and Fig. 2(b). To complete the classes' interface implementation, it is again necessary to add dummy methods. This can result in considerable code size overhead. For example, methods play and play1 are now implemented by all classes, while they were originally only implemented by the Player class.

To enable developers to trade off the number of interfaces merged for the incurred overhead, we merge interfaces in several steps. First, we partition the subtree interfaces into mergeable sets of a tunable size n. Smaller sets will result in merged interfaces with fewer methods, and hence less dummy methods and less overhead. In the second step, each set is merged into a single interface. In the remaining steps, methods are selected and merged to minimize the overhead.

A. Interface partitioning

Besides limiting the overhead, there is another reason for which we cannot merge all interfaces into a single super interface. IM, like CHF, is limited by restrictions imposed by custom class loaders. For example, merging interfaces from different archives can result in linkage errors or class resolution errors from custom class loaders. We therefore limit IM to sets of interfaces of which the subtrees originate from within the same archives. The merged interface can then be packaged in that same archive, such that custom class loaders can find them precisely when and where they need them.

First, we partition all subtree interfaces according to the archives that contain them. Next, each set of interfaces is further partitioned using a first-fit decreasing bin packing strategy [23] that considers the number of classes in a subtree as the size of the corresponding interface. This groups interfaces and their subtrees into a minimal number of bins smaller than or equal to a selected size n. When the interfaces in a bin are later merged, the merged interface will therefore be implemented by at most n classes. For small enough values of n, the resulting packing over similarly sized bins distributes the points-to set sizes in the transformed program evenly.

B. Interface merging

Given a partitioning of all subtree interfaces into n mergeable sets $I_1, \ldots, I_n$. For each set I_j

1) Create a new interface k that declares all methods declared in all $i \in I_j$.
2) Add k to the archive that contains the interfaces in I_j.
3) For each class c in the application, replace all references to $i \in I_j$ by references to k. If c implements an interface $i \in I_j$, make c implement k and add dummy methods to c to complete the implementation of k.
4) Remove all interfaces in I_j from the application.

Fig. 7 shows the resulting hierarchy, with the non-standard d annotation denoting dummy methods. The total number of dummy methods is 59, while there are only 21 non-dummy methods. To limit the overhead of these dummy methods, we merge dummy methods with non-dummy methods in the next steps. In the context of this paper, merging a set of m methods means giving the methods identical signatures and names such that an interface only needs to declare one method instead of m ones, and such that the classes implementing that interface need to implement only one non-dummy method instead of one non-dummy plus $m-1$ dummy methods.

In the hierarchy of Fig. 7, we merge play, decodeSample, decodeFrame, and readFile into merged1, and getStreams and play1 into merged2, as shown in Fig. 3 and in Fig. 2(c) on ll. 2, 5, 20, and 26. Only 19 dummy methods remain.

Method merging (MM) involves merging parameter type lists and return types. Some merged methods will have larger parameter type lists, while some previously void methods now return a value. These changes come at a cost. For instance, for each extra parameter to a method and for each invocation of the method an extra argument has to be passed. To limit these costs and to ensure that program behavior is preserved, MM proceeds in two steps. First, we select sets of related methods that can be merged. Next, we select increasingly more expensive combinations of these sets and merge them greedily.

C. Method set selection

Changing the signature of a method requires making similar changes to the signature of overridden and overriding methods [15]. The MM transformation therefore operates on sets of methods instead of single methods. Each set consists of methods that at all times should have the same signature. Let

- $\mathbb{I}$ be the set of subtree interfaces after CHF or IM, $\mathbb{M}$ the set of all methods, and $\mathbb{C} = \mathbb{A} \cup \mathbb{L}$ the set of all classes.
- $M : \mathbb{C} \mapsto \mathbb{M}$, with $M(c)$ all methods declared in class c.
- $S : \mathbb{M} \mapsto \mathbb{M}^n$ the function that returns the set of methods $S(m)$ that should have the same signature as m.
- $f : \mathbb{M} \mapsto \{false, true\}$, with $f(m)$ indicating whether the signature of m can be rewritten. When it cannot, e.g., because it is referenced via reflection, we can often wrap it in a method of which the signature can be changed.

The method sets that can be merged are then given by $\mathbb{S} = \{S(m) \mid \forall i \in \mathbb{I}\ \forall m \in M(i) \wedge \forall n \in S(m).\, f(n)\}$.

We extend the notions of signature, return type, name, and parameter type list from single methods to sets of methods because each $S \in \mathbb{S}$ is itself a set of same-signature methods.

D. Method set merging

Several constraints limit MM. For example, as a method can only declare one return type, two methods with different non-void return types cannot be merged. Furthermore, we must prevent merging multiple methods with non-dummy implementations and hiding implementations of existing methods by overriding them. While it may seem counter-intuitive that MM in flattened hierarchies can result in methods being overridden, Fig. 5 illustrates that not all subtrees in an application can be flattened. Merging I_2 and I_3 into I results in the hierarchy of Fig. 8. In this hierarchy, dummy methods introduced by merging have been omitted for clarity. The method sets of I:m1 and I:m2 cannot be merged into I:m, because the resulting merged methods E:m and F:m would erroneously hide B:m.

To verify whether or not methods can be merged, we will check for *reaching implementations* in the merged hierarchy. A class c has a reaching implementation of a method m if c or one of its superclasses has an implementation for m. In partially merged interfaces of Fig. 8, methods m1 and m2 could not be merged because classes E and F have a reaching non-dummy implementation for both of them. Based on this observation, we can formalize the *merge condition*: A collection of method sets can be merged iff all sets with a return type other than void have the same non-void return type t, and there is no class that has a reaching non-dummy implementation for two or more methods from different method sets. Based on this condition, we propose the following greedy MM algorithm:

```
while $\mathfrak{S} = \{\{S_i, S_j\} \subset \mathbb{S} \mid \{S_i, S_j\}$ is mergeable$\} \neq \emptyset$
do
    $\mathfrak{s} = \arg\min_{\mathfrak{s}_i \in \mathfrak{S}} C(\mathfrak{s}_i)$
    $\hat{\mathfrak{s}} = \mathrm{merge}(\mathfrak{s})$
    $\mathbb{S} = \mathbb{S} \cup \{\hat{\mathfrak{s}}\} \setminus \mathfrak{s}$
end
```

In this algorithm, $\mathbb{I}$ is the set of merged interfaces obtained after CHF or IM, and $C : \mathbb{S}^* \mapsto \mathbb{R}^+$ is a cost function for which $C(\mathfrak{s})$ gives the cost of merging all sets in $\mathfrak{s}$ as a result of the increase in arguments and the change in return type. The merge subroutine does the actual merging as follows. Given a set $\mathfrak{s} = \{S_i, S_j\} \subset \mathbb{S}$ that adheres to the merge condition

1) Compute signature $s = < r, n, p >$ for merged methods.
2) Create an empty set $\hat{\mathfrak{s}}$ that will hold the merged methods.

	Xvid-Stream	Audio-Stream	Video-Stream	MPGA-Stream	DTS-Stream	MP3-File	Media-File	Media-Stream	MP4-File	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 I: instanceof lookup table

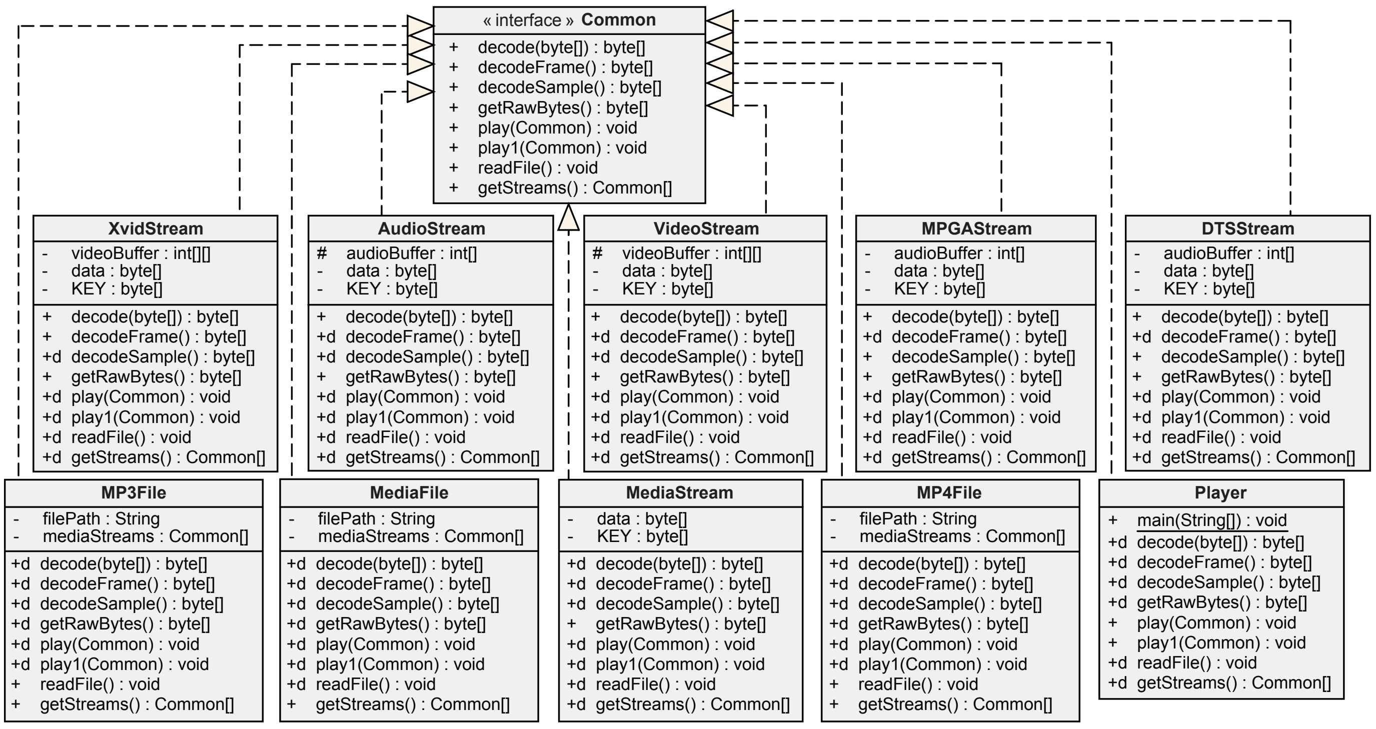

Fig. 7: Class hierarchy of the flattened media player before method merging.

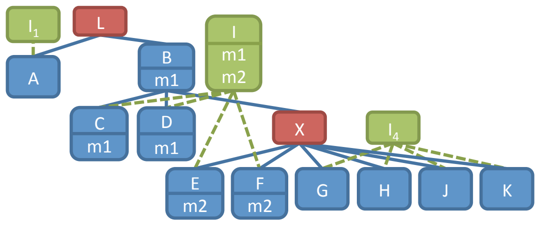

Fig. 8: Class hierarchy after merging I_2 and I_3.

3) Compute for each class c the set $N(c)$ of methods to merge: $N(c) = \{m \in M(c) \mid \exists S_k \in \mathfrak{s} \wedge m \in S_k\}$.
4) For all classes c for which $N(c)$ is not empty:
   a) Create a new method m with signature s.
   b) The body of m becomes the body of the single non-dummy method in $N(c)$ if there is one, or the body of a random method in $N(c)$ otherwise.
   c) Rewrite invocations of methods in $N(c)$ to invocations of m, adding dummy arguments as needed.
   d) Remove all methods in $N(c)$ from c.
   e) Add m to c and to $\hat{\mathfrak{s}}$.
5) Return $\hat{\mathfrak{s}}$.

In step 4a) we compute $s = < r, n, p >$ as follows. The return type r is void if all method sets in $\mathfrak{s}$ return void, else it is the non-void type t of the merge condition. The name n is chosen as a unique random string. The parameter type list p is chosen as a random permutation of the smallest unordered list of types that contains all parameter type lists of the method sets in $\mathfrak{s}$. For example, for two type lists [int, int] and [float], the merged list can be [int, int, float] or any permutation thereof.

V. Obfuscating Factories

Even after IM, some statements still expose type information. For example, after the allocation on l. 9 in Fig. 2(b), player points to an object of type Player. From this information, points-to analyses deduce points-to sets of many local variables. In turn, other attack analyses like call graph construction, program slicing, and type inference will also regain some precision. To prevent this, we replace allocations by calls to object factories [24] that return multiple types of objects. The effect of this object allocation obfuscation, when not undone by an attacker, will be that no points-to analysis, however complicated, will compute more precise results than class hierarchy analysis [25]. To achieve this effect, the transformation proceeds in two steps. First, we preprocess the program to maximize the potency of the transformation. Next, we replace object constructions by calls to object factories.

A. *Code preprocessing*

We want to replace allocations like Common player = new Player() by calls to factories like Common player = MyFactory.create(...). For maximal obfuscation, the factory would return objects of all subtypes of java.lang.Object. However,

```
public void m1() {
  X x1 = new X();
  x1.m0();
  X x2 = new X();
  x2.m1();
  I c = new C();
  c.m2();
  I e = new E();
  e.m1();
}
```

(a) original code

```
public void m1() {
  L x1 = LFactory.create(key1);
  x1.m0();
  B x2 = BFactory.create(key2);
  x2.m1();
  I c = IFactory.create(key3);
  c.m2();
  I e = IFactory.create(key4);
  e.m1();
}
```

(b) code with factories

Fig. 9: Example of object factory insertion.

the return type of a factory must be type-compatible with the variable or field to which the returned object is assigned. In our example, MyFactory.create() has to return an object whose type is equal to or a subtype of Common. To enable the injected factories to return as abstract, information-less types as possible, we first make the local variable types as abstract as possible by means of type inference.

Type inference algorithms use type information from uses and definitions to determine the best suited type for each local variable. Definitions impose a lower bound on a type, uses impose an upper bound. The algorithm by Gagnon *et al.* tries to determine the most concrete possible type for each local variable [26], because more concrete types can aid analyses such as CHA [25] that rely on this type information.

Because more abstract types reduce the available information, we adapted the algorithm to be use-driven instead of definition-driven. When determining a type, we select an upper bound for a local variable based on how it is used. We only use information from definitions to disambiguate between multiple possible upper bounds. For example, assume that method m1 in class B of Fig. 8 is implemented as in Fig. 9(a). The types of x1 and x2 cannot be changed because they are assigned objects of the untransformable type X. Classes C and E are transformable, so the types of c and e have been changed to interface type I. With our type inference, the type of x1 becomes L, because x1 is only used by the invocation of method m0 defined in L. The type of x2 becomes B, because x2 is only used by the invocation of method m1, which is defined in B. The type of c and e does not change, as m1 and m2 are defined in I. Fig. 9(b) shows the resulting code.

B. Object factory creation

We replace object constructions by calls to factories. Let $\mathbb{A}$ be the set of all classes and $\mathbb{I}$ the set of all interfaces of the application. We extend t_s and t^s from $\mathbb{A} \cup \mathbb{L}$ to $\mathbb{U} = \mathbb{A} \cup \mathbb{I}$. For each statement of the form x = new C(...), where $\mathsf{C} \in \mathbb{A}$, and the declared type of x is X, we perform the following steps.

First, we determine the properties of the factory method. We construct $U = \{u \in t_s(X) \cap t^s(C) \mid u \in J_C\}$, where J_C is the jar or archive that contains class C. Given U, we compute the return type of the factory method as

$$R = \underset{u \in U}{\arg\max}\ |q(u)|$$

with $q : \mathbb{U} \rightarrow \mathbb{U}^*$ and $q(u) = \{d \in \mathbb{A} \cap t_s(u) \mid d \in J_C\}$. From R, we compute the set of constructors that will be replaced by and embedded in the factory method as $K = \{m \in M(d) \mid d \in q(R) \wedge m \text{ is a constructor}\}$.

Secondly, given R and K, let RKFactory be the object factory class with a method create with return type R that can create objects using the constructors in K. If this class does not exist yet, we create one in J_C. The parameter type list of create consists of the combined parameter type lists of the constructors in K and one or more extra parameters e. Based on key values that are passed to the factory through e and that identify the original allocation site, the body of the create method invokes the original constructor. The relation between the allocation site, the key passed and the invoked constructor can be hidden behind hashing or white-box crypto.

Finally, replace x = new C(...) by a call to RKFactory.create().

For the example of Fig. 9(a), we create LFactory, BFactory, and IFactory and rewrite allocations as in Fig. 9(b). Without preprocessing, we would have created XFactory to handle the object creations for x1 and x2. That factory would have had the potential to return only seven different types. By contrast, LFactory and BFactory can return 12 and 10 different types, resp. For the media player, Fig. 2(c) shows the resulting code.

VI. Evaluation

We implemented CHF, IM, and OFI in Soot 2.5.0 [27], [28], an analysis and transformation framework for Java bytecode. As our tool rewrites bytecode packaged in a collection of jar files, it does not require access or changes to source code.

Our implementation consists of transformers and a refactoring toolkit. The transformers implement CHF, IM, and OFI as a Soot SceneTransformer, such that they can be applied together with Soot's 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, IM and OFI [14]. In our proof-of-concept tool, the dummy method bodies are empty.

A. Limitations and Restrictions

Clearly, our transformations build on a closed-world assumption, as the whole program to be obfuscated needs to be available for computing points-to sets. To detect the set of non-transformable classes and to ensure that all Java features like 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 [29]. This profile-based tool relies on the whole program to be available and on the developer to provide inputs that generate enough coverage.

Alternatively, a developer can manually complement the coverage of the TamiFlex Play-out Agent with his knowledge of how the program depends on reflection and class loaders. Because we want to evaluate the potential of fully automated type obfuscation, including of third-party software, we chose not to provide any such complementary information.

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

Benchmark	Description	# application classes	# application interfaces	# transformable classes (CHF)	# jar files	code size (MB) pre IO	code size (MB) post IO
avrora	simulates programs on a grid of AVR microcontrollers	1836	83	1657 (90%)	2	4.1	2.9
batik	produces Scalable Vector Graphics using Apache Batik	3787	856	3383 (89%)	7	12.5	9.3
eclipse	executes jdt performance tests for the Eclipse IDE	5213	1261	3886 (75%)	49	25.7	17.2
fop	converts XSL-FI files to PDF	4033	446	3105 (77%)	8	11.0	8.8
h2	executes banking transactions against a database	1843	78	1454 (79%)	5	9.3	7.0
jython	interprets the pybench Python benchmark	3702	166	941 (25%)	8	11.8	10.6
luindex	indexes documents using Lucene	605	28	510 (84%)	4	1.9	1.2
lusearch	performs text searches using Lucene	608	28	510 (84%)	4	1.9	1.2
pmd	analyzes Java classes for source code problems	1999	451	1508 (75%)	7	5.6	4.4
sunflow	renders a set of images using ray tracing	679	59	557 (82%)	3	2.0	1.6
tomcat	executes queries against a Tomcat server	2173	268	1538 (71%)	27	10.1	7.1
xalan	transforms XML documents into HTML	2460	426	2111 (86%)	6	9.6	7.6

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

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.

B. Benchmarks

We used the DaCapo 9.12 benchmark suite [30] to evaluate the protection effectiveness and the performance efficiency of our obfuscations. This suite consists of 14 real-world applications ranging in size from medium to large. We opted for the "9.12 bach" release of the DaCapo suite because the TamiFlex tools have previously been tested on this version (http://dacapobench.org/soot.html). Of the 14 benchmarks, 12 meet the above requirements on class loaders. Their main properties are listed in Table II. Only eclipse and tomcat have enough archives with few classes to have their obfuscation significantly limited at the boundaries of archives. For all but one benchmark the large majority of all classes is transformable. The only exception is jython, 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.

As a baseline for comparison, we use the original DaCapo bytecode, but with identifier names obfuscated [31]. Identifier obfuscation (IO) is orthogonal to CHF, IM, and OFI; any Java obfuscator would apply it. We applied it for our evaluation baseline to present realistic results for obfuscated programs, in particular with respect to code size and memory footprint, both of which heavily depend on the length of identifiers.

From each baseline program, we generated versions with and without CHF, and with and without OFI. On flattened versions, we applied IM at subtree size threshold values of 0 (i.e., no IM), 10, 20, 30, 40 and 50. For each benchmark and each of the 14 combinations of transformations and threshold values, we generated ten different versions with different seeds for the pseudo-random number generators used for bin packing and MM. We report the average results for those ten versions.

C. Provided Protection

Like all obfuscation researchers, we face the problem of measuring our techniques' potency. And as in all of the literature except a few studies involving human subjects, we know of no direct metrics to measure the resistance to reverse-engineering. So instead we rely on established software complexity metrics from the domain of software engineering. Here, we use the static QMOOD metrics from Bansiya et al. [32]. 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 design size [32]. 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 [32], less understandable versions will have lower scores. Fig. 10 displays the relative understandability for our benchmarks after CHF, IM with different thresholds, and OFI. Without OFI, the charts would be almost identical, because OFI does not contribute significantly to the static metrics considered in QMOOD. The charts show that CHF and IM do reduce QMOOD understandability significantly, with understandability dropping as more interfaces are merged. Overall, there is little correlation between the QMOOD result and the benchmark properties of Table II. For jython, with only 25% of its classes transformed, the result is comparable to benchmarks that have more than 70% of their classes transformed. This illustrates the limitations of QMOOD.

As a representative sample, Fig. 11 shows the relative contribution of the different QMOOD metrics to lusearch's understandability, for the same program versions as in Fig. 10. Positive/negative contributions mean that higher/lower values of a metric contribute to lower understandability. In the original programs, all metrics contribute $\pm 33\%$. In combination with the results for lusearch in Fig. 10, this chart shows that most of the understandability reduction results from increases in the amounts of coupling, polymorphism and complexity as defined in QMOOD. As more IM is applied, the growing QMOOD complexity, which basically equals the growing number of (dummy) methods, becomes more dominant. Since the dummy methods are empty in our current implementation, the observed increase in QMOOD complexity does not reflect a real complexity increase, however. So also in this regard, we hit an important limitation of QMOOD. In summary, while

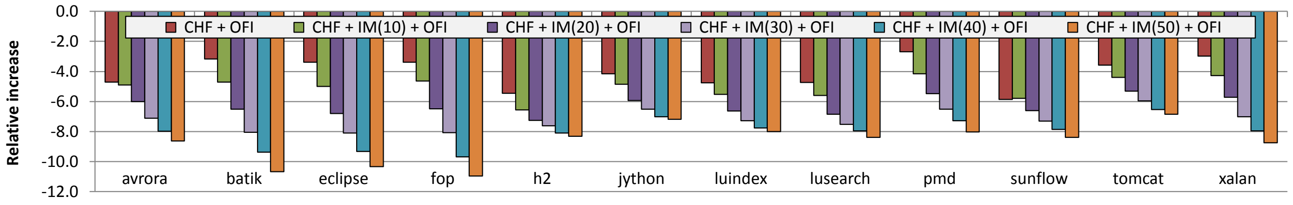

Fig. 10: QMOOD understandability

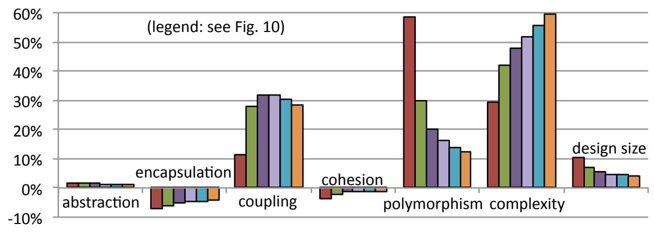

Fig. 11: Contribution of QMOOD metrics for lusearch

QMOOD metrics hint that our obfuscations provide some real protection, QMOOD clearly needs to be combined with other metrics before we can draw more conclusions.

To complement QMOOD, we evaluate the obfuscations' ability to confuse static analyses. In practice, the precision of many important client analyses, including call graph construction and virtual call resolution, can drop significantly as the result of an imprecise points-to analysis. At the same time, the memory footprint and execution time of those analyses increase with less precise points-to analysis because the constructed call graphs become bigger. Hence, reducing the precision of points-to analysis by causing it to return larger sets will directly reduce the effectiveness and efficiency of several static analyses that are fundamental for static attacks.

For this part of our evaluation, we relied on the more robust and configurable T.J. Watson Libraries for Analysis (WALA, http://wala.sf.net) to compute points-to sets using simple class hierarchy analysis [25], context-insensitive 0-CFA [18], and the partially context-sensitive, 0-1-CFA and 0-1-container-CFA of WALA. Because of space concerns we only report results obtained with 0-1-container-CFA, the strongest of the four. To ensure that WALA's call graph construction includes the program parts that are reachable through reflection, we ran WALA on benchmark versions in which TamiFlex Booster had replaced indirections through reflection by direct invocations [29]. Fig. 12 presents the average points-to set sizes for all local variables and parameters of methods for the different benchmark versions. Four key observations are to be made.

First, many points-to sets in jython are huge because of its dynamic class generation. Its presence devastates precision of the analysis in large parts of the program. Many points-to sets become so large they are not meaningful anymore. CHF, IM, and OFI can then not damage the analysis any further.

Secondly, the leftmost light bar of several other benchmarks shows that the points-to sets do not always grow when only OFI is applied. This results from the fact that the declared return types of the inserted factories have very few subtypes in the original class hierarchy. As such, they cannot obfuscate a lot of type information. After CHF and aggressive IM, by contrast, OFI always increases the points-to set sizes.

Thirdly, on some benchmarks CHF applied in isolation reduces the average points-to set sizes. This is due to the this pointers in methods of flattened classes, which by construction have singleton points-to sets. As the methods' first implicit parameter, this pointers contribute to the computed average points-to set sizes. Their negative effect is even more pronounced because the inserted getters and setters are very small methods that only contribute their this pointer but no local variables or other parameters. A similar effect plays when more aggressive IM is applied. In that case parameters added during MM contribute very small points-to sets that bring down the average sizes. Depending on the benchmark, CHF and/or IM can or cannot obfuscate enough type information in other places of the programs to compensate the effects of the this pointers and of added parameters.

Fourthly, without OFI, even aggressive IM typically does not make the points-to set grow. This is because WALA's advanced analysis can extract and propagate a lot of type information from allocation sites. Only the obfuscation of that information by OFI makes the points-to sets grow. The light bars growing from left to right for each benchmark indicate that OFI becomes more effective with more aggressive IM.

Combined, CHF, IM, and OFI increase the average points-to set sizes (excl. jython) with a factor 4.67 on average, ranging from a factor 1.78 for batik to a factor 11.98 for luindex.

D. Overhead

The dark bars in Fig. 13(a) show how code size grows with more obfuscation. The lighter bars on top indicate the code size saved by means of MM, i.e., what the size would be without MM. As expected, more IM implies more code. The increase in code size varies, but overall, the price of the obfuscations is quite large. For most benchmarks, MM works well. The only exception is eclipse, where the unbounded merging of parameter lists in our current implementation introduces more overhead than the merging of methods actually saves.

The run-time overheads reported in Fig. 13(b,c) include all 10 runs of the benchmarks in their harness. This includes the warm-up runs during which the JIT compiler is active. Fig. 13(b) depicts the relative total number of bytes allocated on the heap by the different program versions. As objects do not grow in size because of our obfuscations, the obfuscated programs require very little additional heap memory. When more is needed, this mainly results from class loaders now loading bigger class files. For pmd, the increase is caused by the program analyzing its own bytecode by means of the included ASM library (http://asm.ow2.org). Because the

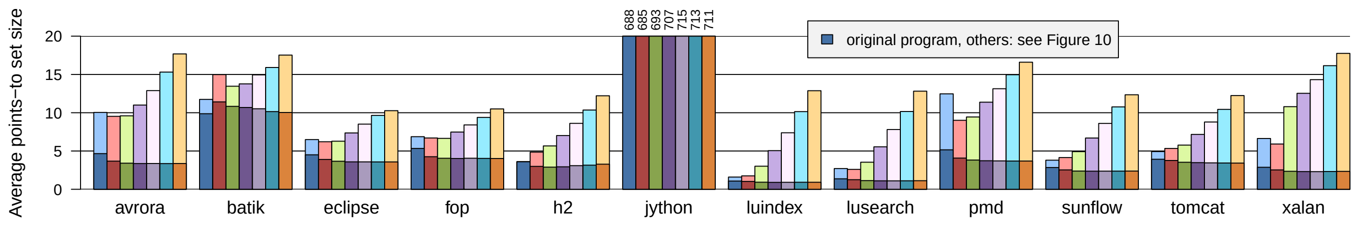

Fig. 12: Average points-to set sizes. Dark bars denote sizes obtained without OFI, light bars denote sizes with OFI.

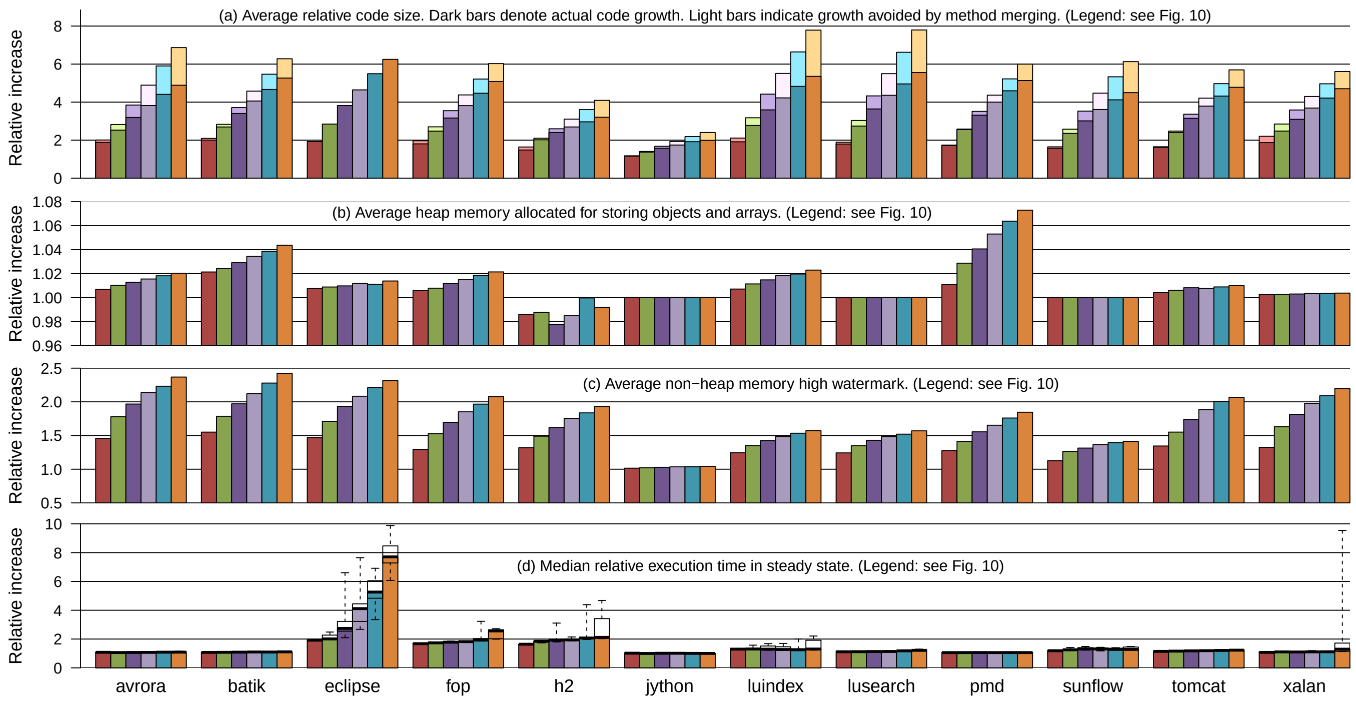

Fig. 13: Overhead of the obfuscations. All metrics are relative to the original benchmarks.

program classes have grown, much more objects are allocated on the heap for ASM's internal bytecode representation. A program taking itself as input in this way obviously constitutes very atypical behavior, so its heap overhead is not representative of the overhead on other programs.

For some benchmarks, the amount of data allocated on the heap does not increase monotonically with the amount of IM. This is due to the sampling-based JIT compiler behaving differently on different benchmark versions. Experiments with JIT optimization levels vs. interpreted execution revealed that less data is allocated on the heap when more aggressive JIT compilation is used, because escape analysis enables the allocation of data on the stack instead of the heap [33]. For some benchmarks, like h2, the escape analysis accidentally performs better on some more heavily obfuscated versions.

Fig. 13(c) depicts overhead in terms of the run-time non-heap memory high-watermark. The non-heap memory is mainly used to store code. The overhead is hence proportional to the code size overhead. On average, the non-heap memory is many times smaller than the heap. The run-time memory footprint overhead of our obfuscations is hence limited.

In contrast with the previously discussed levels of overhead and obfuscations, which do not depend on random seeds used, we observed that the performance overhead can vary significantly within a set of 10 benchmark versions generated with identical IM thresholds, but with different random seeds. This can be seen in Fig. 13(d), which shows the measured steady-state performance overhead using standard box plots. For most benchmarks, the performance overhead is very limited. For eclipse, however, the median slowdown is 670%. For this benchmark, additional measurements revealed that CHF is responsible for 20–27 percentage points (pp), mainly because of our instanceof replacement and to some extent also because of the introduced getters and setters. As a flattened hierarchy contains much more getter and setter implementations than an unflattened one, they are inlined less efficiently by the JIT compiler. OFI causes 56–393 pp of the slowdown. This grows so much with increasing IM, because as factories become more generic, they incorporate more constructors, of which all parameter type lists are merged into the factories' parameter lists. The slowdown results from having to pass values for all those parameters at all factory calls. Similarly, more IM implies more MM, and hence also longer parameter lists, resulting in an additional overhead of up to 250 pp.

From the variation in overhead observed for some versions of eclipse, fop, h2, and xalan that were generated with different random seeds but that feature similar levels of obfuscation, as well as from the fact that the unbounded application of MM did not benefit eclipse's code size, we learn that there is a lot of potential for reducing the overhead of our obfuscations by making the currently partially randomized and unbounded application of merging more tuned, controlled, and limited.

VII. Correctness, reliability and maintainability

A formal validation of CHF, IM, and OFI is out of the scope of this paper. Several observations can, however, increase confidence in their soundness and preservation of program semantics. First of all, we checked that all benchmark versions passed type verification and produced correct output. These checks were not limited to the 12x14x10 versions reported in the previous section. We also checked many other versions generated while studying alternative heuristics for IM and MM, some of which were reported in our previous work [34], as well as intermediate program versions generated by the individual transformation steps discussed in sections III, IV, and V. All of these experiments were stressing Soot, WALA, and TamiFlex beyond their pre-existing capabilities, so we had to fix numerous bugs in these tools. One debugging technique consisted of comparing the constructed call graphs and points-to sets of the programs boosted with TamiFlex before and after the obfuscations. Once we had fixed all bugs, we verified that the graphs and sets obtained after obfuscation completely cover the graphs and sets before obfuscations.

Regarding the TamiFlex Play-out Agent and Booster, we point to the literature for a discussion of their validity [29].

As mentioned in Section VI, our obfuscations are quite similar to existing refactorings [14], [15], [35]. Tip et al. express the valid refactoring space by means of type constraints [15]. Based on the original program's code and points-to sets, a set of type constraints is constructed that constrain the types that can be used in the program's declaration. These constraints determine 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 shadowing and 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, incl. 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, the class loader restrictions imposed in Section VI-A allow us to 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 [36] remain intact.

To facilitate further validation and testing, researchers can obtain our source code and benchmarks upon simple request.

Besides providing obfuscation and introducing overhead, our transformations come with some important side effects.

User bug reports on obfuscated programs are harder to interpret. However, since there is a 1-to-m mapping between the classes, methods and fields in the original program and those in the obfuscated program, it is straightforward to translate back traces from the obfuscated to the original code.

As our transformations alter the execution speed of different code fragments differently, they may expose or hide race conditions in multithreaded code. In this regard, our transformations do not differ from any other static or JIT code optimization, or any virtual machine tuning.

Finally, our obfuscations have limited impact on maintainability. Being applied on the bytecode after testing and right before the code is distributed to customers, the obfuscations do not affect the source developer directly. However, since the obfuscations build on whole-program analysis, simple patches in one class' source code require the reapplication of the obfuscation to the whole program, which may well result in changes to most of the obfuscated bytecode. So distributing updates might require more bandwidth.

VIII. Related work

Compared to a preliminary presentation of CHF [34], this paper presented results obtained with a more complete, mature prototype tool. For example, the tool used here was able to flatten 75% of the classes in eclipse, compared to 38% before. We also evaluated more benchmarks, and used more advanced points-to analyses. This explains the sometimes different results presented here. Whereas the preliminary report only included the notion of IM with a very preliminary implementation, this paper combined CHF with an optimized IM process that includes method merging, as well as OFI.

As far as we know, we are the first to automate class hierarchy obfuscations in a tool that can handle complex applications that heavily use reflection and custom class loading.

To obfuscate an application's design, its class hierarchy and the type information contained in its 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 [13]. 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 become 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 results in a much larger memory footprint. The authors acknowledge this potential issue, but their experimental evaluation is limited to execution time measurements of relatively small and simple programs (up to 307 classes). For those, they measure slow-downs up to 130% even with limited coalescing.

CHF can be combined with class coalescing. In particular, CHF enables more efficient coalescing. Coalescing MP3File and VideoStream in Fig. 1 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. After CHF, MP3File and VideoStream can be coalesced without affecting the size of objects of other classes.

The false factoring transformation by Collberg et al. [4] 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. CHF can prepare a program for false factoring [4]. In Fig. 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.

Given a set of transformable classes, the obfuscation techniques introduced by Sakabe et al. [10] first change 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.

Like Sakabe et al., we use interfaces as common super types. To limit the number of methods in these interfaces, their approach requires the use of special parameter and return objects that have to be created for each method call and return. Because this can result in large run-time overheads, we instead use method merging to make method signatures more uniform. This generally has a much smaller performance impact. Our OFI also differs from their type hiding transformation. First, we use factory methods rather than inline code to create new objects. These factories can become larger and more complex without inflating the code too much. Additionally, because of our custom type inference, each factory we create can return the maximum number of possible types of objects.

Snelting and Tip [37], [38] presented a method for analyzing and re-engineering 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 [35]. 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.

Another attack approach could build on diffing tools such as Stigmata (http://stigmata.sourceforge.jp). Such tools can assist in inferring the original class hierarchy by identifying duplicated methods and fields in the flattened classes. To distract such tools, we could introduce artificial differences or similarities by choosing appropriate dummy method bodies.

So far, this section focused on related work that involves class hierarchy transformations. That work, like CHF, IM, and OFI has little in common with the decompilation, identifier, data flow, and control flow obfuscations mentioned in Section I. In fact, the different types of obfuscation are mostly complementary. CHF, IM, and OFI do not hinder decompilation in any way, and neither do they aim for getting nasty decompiled code. They only aim for bytecode (and corresponding decompiled source code) that provides as little as possible static type information. Moreover, we combined them with identifier obfuscation for our experimental evaluation. Still, it is noteworthy that our transformations' resulting larger points-to sets and larger call graphs likely open up opportunities for alias-based obfuscations [4], [6], [31], [39].

IX. Conclusions and Future Work

With CHF, IM, and OFI, we presented three obfuscations for object-oriented programs written in managed programming languages. With a prototype implementation, we obfuscated real-world Java programs that feature reflection and custom class loading. The experiments demonstrated that all three transformations are needed in order to achieve good results. Combined, they provide measurable protection against both human understandability and automated program analysis. QMOOD understandability decreased with factors 7–11, and average points-to set sizes increased with factors 2–12.

These obfuscations were typically, but not always, obtained with low performance and memory footprint overhead, but at a significant code size overhead of up to a factor 6. A simple method to trade off overhead for protection is available, as one can easily tune the number of interfaces merged during IM.

We envision future work in two directions: stronger protection and reduced overhead. First, we see a lot of potential in alternative IM strategies that do not focus on filling bins but instead focus on maximal obfuscation. Such strategies could, e.g., try to estimate the effect of merging different interface combinations on points-to set sizes. Alternatively, IM could be driven by a developer's categorization of more and less sensitive code portions. Furthermore, the larger points-to sets and larger call graphs obtained after our transformations open up opportunities for alias-based program obfuscations. Different combinations of such techniques should be explored.

The large variations in performance overhead observed for different versions of some benchmarks with similar levels of obfuscation, all of which were generated with a partially randomized decision logic, and the lack of code size improvement through MM for one benchmark, suggest a potential for reducing our obfuscations' overhead. We plan to make IM cost-aware by taking into account MM opportunities during IM, rather than considering MM an afterthought as in our current implementation. Furthermore, machine-learning techniques and profile information will be useful to limit the application of MM and OFI in situations where the resulting overhead as a result of parameter list merging becomes too high without contributing much to the achieved obfuscation.

References

[1] S. Holst, "Assessing and managing security risks unique to Java and .NET," *ISSA J.*, 2009.

[2] J.-T. Chan and W. Yang, "Advanced obfuscation techniques for Java bytecode," *J. of Syst. and Software*, vol. 71, no. 1-2, pp. 1–10, 2004.

[3] T. Hou *et al.*, "Three control flow obfuscation methods for Java software," *IEE Proc.-Software*, vol. 153, no. 2, pp. 80–86, Apr 2006.

[4] C. Collberg *et al.*, "A taxonomy of obfuscating transformations," University of Auckland, Tech. Rep., 1997.

[5] Y. Zhou *et al.*, "Information hiding in software with mixed boolean-arithmetic transforms," in *Proc. Int. Conf. on Inf. Security Applicat.*, 2007, pp. 61–75.

[6] C. Collberg *et al.*, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proc. ACM SIGPLAN-SIGACT Symp. on Principles of Program. Lang.*, 1998, pp. 184–196.

[7] A. Majumdar and C. Thomborson, "Manufacturing opaque predicates in distributed systems for code obfuscation," in *Proc. Australasian Comput. Sci. Conf.*, 2006, pp. 187–196.

[8] A. Majumdar *et al.*, "A survey of control-flow obfuscations," in *Int. Conf. on Inf. Syst. Security*, 2006, pp. 353–356.

[9] J. Palsberg *et al.*, "Experience with software watermarking," in *Proc. Annu. Computer Security Applicat. Conf.*, 2000, pp. 308–316.

[10] Y. Sakabe *et al.*, "Java obfuscation approaches to construct tamper-resistant object-oriented programs," *IPSJ Digital Courier*, vol. 1, pp. 349–361, 2005.

[11] A. P. R. Venkatraj, "Program obfuscation," Master's thesis, University of Arizona, 2003.

[12] M. Batchelder and L. Hendren, "Obfuscating Java: the most pain for the least gain," in *Proc. Int. Conf. on Compiler Constr.*, 2007, pp. 96–110.

[13] M. Sosonkin *et al.*, "Obfuscation of design intent in object-oriented applications," in *Proc. ACM Workshop on Digital Rights Management*, 2003, pp. 142–153.

[14] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[15] F. Tip *et al.*, "Refactoring using type constraints," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 3, pp. 9:1–9:47, 2011.

[16] B. Bellamy *et al.*, "Efficient local type inference," in *Proc. ACM SIGPLAN Conf. on Object-Oriented Program. Syst. Lang. and Applicat.*, 2008, pp. 475–492.

[17] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for Java," in *Proc. ACM SIGPLAN Conf. on Program. Lang. Design and Implementation*, 2006, pp. 387–400.

[18] D. Grove *et al.*, "Call graph construction in object-oriented languages," in *Proc. ACM SIGPLAN Conf. on Object-Oriented Program., Syst., Lang., and Applicat.*, 1997, pp. 108–124.

[19] M. Hind and A. Pioli, "Evaluating the effectiveness of pointer alias analyses." *Sci. of Comput. Program.*, vol. 39, no. 1, pp. 31–55, 2001.

[20] B. G. Ryder, "Dimensions of precision in reference analysis of object-oriented programming languages," in *Proc. Int. Conf. on Compiler Construction*, Warsaw, Poland, 2003, pp. 126–137.

[21] E. McCluskey, *Introduction to the theory of switching circuits*. McGraw Hill Text, 1965.

[22] S. Chow *et al.*, "White-box cryptography and an AES implementation," in *Revised Papers Int. Workshop on Selected Areas in Cryptography*, 2003, pp. 250–270.

[23] D. S. Johnson, "Near-optimal bin packing algorithms," Ph.D. dissertation, Massachusetts Institute of Technology, 1973.

[24] E. Gamma *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[25] J. Dean *et al.*, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proc. European Conf. on Object-Oriented Program.*, 1995, pp. 77–101.

[26] E. Gagnon *et al.*, "Efficient inference of static types for Java bytecode," in *Proc. Int. Symp. on Static Analysis*. Springer, 2000, pp. 199–219.

[27] P. Lam *et al.*, "The Soot framework for Java program analysis: a retrospective," in *Cetus Users and Compiler Infrastructure Workshop*, Oct. 2011.

[28] R. Vallée-Rai *et al.*, "Soot - a Java bytecode optimization framework," in *Proc. Conf. of the Centre for Adv. Stud. on Collaborative Research*, 1999, pp. 125–135.

[29] E. Bodden *et al.*, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proc. Int. Conf. on Software Eng.*, 2011, pp. 241–250.

[30] S. M. Blackburn *et al.*, "Wake up and smell the coffee: evaluation methodology for the 21st century," *Communications of the ACM*, vol. 51, no. 8, pp. 83–89, Aug. 2008.

[31] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.

[32] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002.

[33] P. Molnar, A. Krall, and F. Brandner, "Stack allocation of objects in the CACAO virtual machine," in *Proc. Int. Conf. on Principles and Practice of Programming in Java*, 2009, pp. 153–161.

[34] C. Foket *et al.*, "A novel obfuscation: class hierarchy flattening," in *Proc. Int. Symp. on Foundations and Practice of Security*, 2012, pp. 194–210.

[35] M. Streckenbach and G. Snelting, "Refactoring class hierarchies with KABA," in *Proc. ACM SIGPLAN Conf. on Object-Oriented Program., Syst., Lang., and Applicat.*, 2004, pp. 315–330.

[36] L. Gong, *Java^{TM}SE Platform Security Architecture*, [Online] Available: http://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-spec.doc.html.

[37] G. Snelting and F. Tip, "Reengineering class hierarchies using concept analysis," in *Proc. ACM SIGSOFT Int. Symp. on Foundations of Software Eng.*, 1998, pp. 99–110.

[38] ——, "Understanding class hierarchies using concept analysis," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 3, pp. 540–582, May 2000.

[39] A. Majumdar *et al.*, "On evaluating obfuscatory strength of alias-based transforms using static analysis," in *Proc. Int. Conf. Advanced Computing and Communications*, 2006, pp. 605–610.

Christophe Foket is a Ph.D. student at Ghent University in the Computer Systems Lab. He obtained his Bsc. degree in Informatics from Ghent University's Faculty of Sciences in 2007 and his Msc. degree in Computer Science from Ghent University's Faculty of Engineering in 2009. His research focuses on obfuscation of bytecode applications to defend against reverse engineering and code lifting attacks.

Bjorn De Sutter is a professor at Ghent University in the Computer Systems Lab. He obtained his Msc. and Ph.D. degrees in Computer Science from Ghent University's Faculty of Engineering in 1997 and 2002. His research focuses on the use of compiler techniques to aid programmers with non-functional aspects of their software, such as performance, code size, reliability, and software protection.

Koen De Bosschere is professor at the Engineering School of Ghent University, Belgium where he teaches courses on computer architecture and operating systems. His research interests are binary translation, virtualization, and software protection. He authored and co-authored over 170 peer-reviewed papers. He is the coordinator of HiPEAC, the European Network of Excellence on High Performance and Embedded Architecture and Compilation and of the yearly ACACES summer school.