Globale obfuscatie van bytecode-toepassingen

Global Obfuscation of Bytecode Applications

Christophe Foket

Promotoren: prof. dr. ir. B. De Sutter, prof. dr. ir. K. De Bosschere
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. R. Van de Walle
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2015 - 2016

UNIVERSITEIT GENT

ISBN 978-90-8578-860-7
NUR 980
Wettelijk depot: D/2015/10.500/104

Dankwoord

“Zou jij willen doctoreren?” … doctoreren … Je de komende zoveel jaar van je leven toeleggen op één specifiek onderwerp, daarin een expert worden, nieuwe technieken ontwikkelen, daarover papers publiceren en ze presenteren op internationale conferenties. Het klonk als een zware taak toen ik van Bjorn die vraag kreeg. Begrijp me niet verkeerd, dat was het ook. Gelukkig heb ik op veel hulp en steun kunnen rekenen van talloze personen die ik hieronder zou willen bedanken.

Eerst en vooral zou ik mijn promotoren Bjorn en Koen willen bedanken om mij de kans te geven om te doctoreren en mij al die jaren bij te staan in mijn onderzoek. Zonder hun begeleiding en hun kritische maar hulpvolle opmerkingen zowel tijdens mijn onderzoek als tijdens het schrijven was dit werk er nooit gekomen.

Furthermore, I would also like to thank the other members of my exam committee: prof. Eric Bodden, prof. Christian Collberg, prof. Bart Dhoedt, dr. Eric Lafortune, and prof. Hendrik Van Landeghem. Their constructive criticism helped improve this final version of my dissertation.

Daarnaast zou ik graag Universiteit Gent en het agentschap voor Innovatie door Wetenschap en Technologie (IWT) willen bedanken voor hun financiële steun. Verder uit ik ook graag mijn dank aan Universiteit Gent, het Vlaams Supercomputer Centrum (VSC), de Herculesstichting en de Vlaamse Overheid – departement EWI om de benodigde rekenkracht voor de experimenten in dit werk te verschaffen. Voor alle andere technische en logistieke ondersteuning gaat mijn dank uit naar Karen, Marnix, Michiel, Ronny, Annelies en Rita.

De afgelopen jaren heb ik met veel verschillende collega’s een kantoor gedeeld. Bart, Jonas, Jeroen, Niels, Panagiotis, Stijn, Wim, Ronald, Tim, Bert, Jens, Hadi, Peng, Sander, Henri, Farhadur, het was een plezier om met jullie in hetzelfde team te mogen werken. Ik heb van jullie ontzettend veel bijgeleerd over zaken waarvan ik niet eens wist dat ze bestonden.

Door al die jaren met verschillende collega's een kantoor te delen ontstaan er uiteraard vriendschappen, waardoor er ook buiten het werk met "collega's" wordt afgesproken. Mooie momenten waren hierbij de wekelijkse pizza's in de Prima Donna, de robotcompetitie van WELEK waar ik samen met Panagiotis en Wim meermaals aan heb deelgenomen, de semi-regelmatige movie nights, de vele honderden bowling-games met Jeroen en Stijn, en uiteraard het bezoek aan Black Hat en DEFCON en onze aansluitende trip door de VS in de zomer van 2012.

Verder zou ik ook de vaste klanten van Overpoort Bowl willen bedanken om altijd wel in de stemming te zijn om even stoom af te blazen na een drukke werkdag. Ook het vaste personeel mag zeker niet ontbreken. Arne, Dany, Jos, Stef, bedankt om voor mij altijd wel een vrije bowlingbaan te regelen, zelfs op de drukste momenten.

Ten slotte zou ik ook mijn familie van harte willen bedanken voor hun steun de afgelopen jaren. Ma, Pa, Tasha, Meme, Pepe, Guido, Christine, bedankt om altijd in mij te blijven geloven.

Christophe Foket
Gent, 13 november 2015

Examencommissie

Leescommissie

Samenvatting

Technieken en tools om programma's te reverse engineeren en aan te passen zijn veel softwareontwikkelaars niet vreemd. Kwaadwillige gebruikers kunnen ze echter gebruiken om software aan te vallen met verschillende doeleinden, waaronder softwarepiraterij, malware-injectie en diefstal van de intellectuele eigendom die ingebed is in de software. Om hun software te beschermen tegen reverse engineering en geautomatiseerde aanvallen kunnen softwareleveranciers zich wenden tot obfuscatietechnieken. Deze technieken transformeren programma's om ze complexer te proberen maken, met als doel hun interne werking te verbergen, potentiële aanvallers af te wenden en aanvalstools te verwarren. Zo zullen controleobfuscaties bijvoorbeeld proberen om het verloop van een programma te verbergen achter vals controleverloop, terwijl dataobfuscaties zullen proberen om de werkelijke gegevens waarop een programma berekeningen uitvoert te verbergen.

Controle- en dataobfuscaties volstaan in vele gevallen om binaire programma's te transformeren zodat het moeilijker wordt om deze te analyseren. Voor bytecode-programma's is dit echter niet het geval. Bytecode-programma's bevatten namelijk veel meer meta-informatie in de vorm van typehiërarchieën, casts, declaraties van velden en signaturen van methoden. Deze overvloed aan meta-informatie vergemakkelijkt niet alleen just-in-time-compilatie en bytecode-verificatie; ze kan ook gebruikt worden door potentiële aanvallers.

Om de meta-data die in bytecode-toepassingen is ingebed te obfusceren hebben onderzoekers reeds verschillende technieken voorgesteld om, onder andere, de namen van klassen en hun methoden en velden te vervangen door nietszeggende, willekeurig gegenereerde namen. Anderen hebben dan weer voorgesteld om het ontwerp van applicaties te verbergen door klassen te splitsen of samen te voegen, of door valse overervingsrelaties te creëren tussen ongerelateerde klassen, waardoor

het lijkt alsof deze verwant zijn.

In dit doctoraat gaan we een stap verder in het obfusceren van het ontwerp van applicaties. We presenteren drie complementaire obfuscatietechnieken voor Java bytecode-programma's. Gecombineerd obfusceren deze technieken een belangrijk onderdeel van het ontwerp van een applicatie en veel van de aanwezige type-informatie. Onze eerste transformatie, *class hierarchy flattening* (CHF), heeft als doel de klassenhiërarchie van een programma maximaal af te vlakken door de subtyperelaties tussen de klassen zoveel mogelijk te verbreken. Zodoende zal CHF al een deel van de type-informatie mee verwijderen, maar desondanks zal een aanvaller toch nog veel informatie kunnen afleiden uit de types die toegewezen zijn aan velden en methodeparameters, en uit de types die gebruikt worden in casts en bij het aanmaken van nieuwe objecten. Om ook deze informatie te verbergen presenteren we twee extra transformaties: *interface merging* (IM) en *object factory insertion* (OFI).

Interface merging vermindert de hoeveelheid type-informatie in een programma door meerdere interfaces samen te voegen tot één interface. Op die manier implementeren meer klassen dezelfde interface, waardoor variabelen en velden wiens types interfacetypes zijn naar meer verschillende types objecten kunnen verwijzen. Object factory insertion vervangt code die nieuwe objecten aanmaakt door code die zogenaamde factory-methoden oproept. Deze methoden kunnen op hun beurt veel verschillende constructors kunnen oproepen en zo veel verschillende types objecten teruggeven. In combinatie met CHF en IM zorgt OFI ervoor dat het moeilijk wordt om het exacte type van het object te bepalen waarnaar velden, variabelen of parameters verwijzen op verschillende plaatsen in het programma, zonder het programma effectief uit te voeren. Ten gevolge hiervan zal niet alleen een aanvaller het moeilijker hebben om een geobfusceerd programma te begrijpen; ook de prestatie van zijn tools die steunen op type-informatie zal hier onder lijden.

Om na te gaan hoeveel bescherming onze transformaties bieden en hoeveel overhead ze teweegbrengen, hebben we ze geïmplementeerd in een obfuscator, en daarmee programma's uit de *DaCapo benchmark suite* geobfusceerd. Op basis van complexiteitsmetrieken uit het software engineering-domein tonen we aan dat onze obfuscaties programma's minder verstaanbaar maken voor een menselijke aanvaller en dat programma's tevens steeds complexer worden naarmate onze obfuscaties agressiever worden toegepast. Daarnaast tonen wij ook aan dat onze technieken het aantal types objecten waarnaar velden, variabelen en argumenten kunnen verwijzen tot twaalf keer in grootte doen toenemen, waardoor ze de prestatie van statische analyse-tools die gebruik maken van precieze type-informatie sterk verminderen.

Net als vele andere obfuscatietransformaties kunnen ook de onze een aanzienlijke kost met zich meebrengen. Voor de meeste testprogramma's is de vertraging en de extra hoeveelheid geheugen dat nodig is om het programma uit te voeren beperkt. Sommige sterk geobfusceerde versies van één van onze testprogramma's werden echter meer dan acht keer trager dan het origineel. Verder maten we voor bijna alle testprogramma's een grote toename in de grootte van hun geobfusceerde versies. In sommige gevallen liep die toename zelfs op tot 700%.

Onze experimenten toonden aan dat de grootste toename in uitvoeringstijd veroorzaakt werd door object factory insertion, terwijl de grootste toename in applicatiegrootte te wijten was aan method merging (MM), een techniek die we ontwikkeld hebben om de toename in applicatiegrootte ten gevolge van class hierarchy flattening en interface merging te verminderen. Zowel OFI als MM combineren de signaturen van verschillende methoden, zij het om factory-methoden te construeren die verschillende constructors kunnen oproepen, of om methoden samen te voegen. In beide gevallen kunnen de parametertypelijsten van methoden zodanig lang worden dat zij voor veel overhead zorgen. In het geval van OFI zorgt het grote aantal parameters ervoor dat het oproepen van een factory-methode vele malen duurder is dan het oproepen van de individuele constructors, gezien veel meer argumenten moeten meegegeven worden telkens een object moet worden aangemaakt. In het geval van MM kunnen de samengevoegde parametertypelijsten zodanig lang worden dat meer ruimte nodig is om deze lijsten op te slaan als deel van de meta-informatie in het programma, dan dat er ruimte wordt vrijgemaakt door methoden samen te voegen.

Om de overhead van deze transformaties te verminderen, hebben we ons gericht op een aantal van hun gebreken. We hebben verschillende verbeteringen aangebracht aan het method merging-algoritme en het uitgebreid met een model dat de impact van verschillende merge-operaties op de applicatiegrootte nauwkeurig kan inschatten. Op die manier kan het algoritme steeds die methoden samenvoegen die zorgen voor de grootste vermindering in applicatiegrootte. Een ander voordeel dat verbonden is aan het gebruik van dit model is dat het algoritme nu ook eenvoudig kan stoppen met het samenvoegen van methoden indien het merkt dat de grootte van de applicatie anders zou toenemen.

Verder hebben we OFI aangepast zodat deze transformatie constructors nu verdeelt over verschillende factory-methoden op basis van hoe vaak elke constructor wordt uitgevoerd, in plaats van alle constructors onder te brengen in één factory-methode. Op die manier kunnen vaak uitgevoerde constructors gegroepeerd worden in factory-methoden met weinig argumenten, om zo de overhead in uitvoeringstijd te beperken.

Bijkomende experimenten tonen aan dat onze technieken door bovenvermelde verbeteringen weinig moeten inboeten op vlak van beveiliging, maar dat de overhead van geobfusceerde programma's wel veel lager is. Gemiddeld daalt de overhead in programmagrootte met meer dan 30%, terwijl de overhead in uitvoeringstijd met meer dan 40% daalt. Ten gevolge van deze dalingen kunnen onze verbeterde technieken, in vergelijking met hun originele varianten, niet alleen gebruikt worden om vergelijkbare bescherming aan te bieden in ruil voor minder overhead; ze kunnen ook gebruikt worden om betere bescherming aanbieden in ruil voor dezelfde overhead.

Summary

Reverse engineering and modification of software are well-understood and common practices. Malicious users can use them to attack software with the goals of software piracy, software IP theft, data theft, and malware injection. To protect their software against reverse engineering attempts and automated attacks, software vendors may turn to obfuscation techniques. These techniques transform programs in an attempt to make them appear more complex, to make it more difficult to extract certain parts of information from them, to confuse potential attackers, and to reduce the effectiveness of attack tools. Control obfuscations, for instance, try to hide the actual flow of the application by introducing spurious control flow, while data obfuscations try to hide the actual data values a program operates on.

In many cases control and data obfuscations are sufficient to make binary applications more difficult to analyze. However, for bytecode applications they usually fall short. Bytecode applications inherently contain much more meta-information, which they expose through type hierarchies, casts, field declarations and method signatures. This abundance of meta-information not only facilitates just-in-time (JIT) compilation and bytecode verification; it also helps potential attackers.

To obfuscate the meta-data embedded in bytecode applications, several researchers have suggested techniques to replace the names of classes and their members with randomly generated ones. Others have suggested to obfuscate the design of the application by splitting or merging classes, or by introducing fake inheritance relations between unrelated classes to make it seems as if they are related.

In this work, we take design obfuscation one step further. We present three complementary obfuscation techniques for Java bytecode applications that, when used in combination, obfuscate an important part of an application's design, and much of its type information. Our first transformation, *class hierarchy flattening* tries to get rid of an application's class hierarchy by maximally removing the subtype relations between its classes. In doing so, it already removes some type information. However, even after flattening, programs generally still contain much type information in the form of field and method signatures, casts, and object creation expressions. To also hide this information, we present two additional transformations: *interface merging* (IM) and *object factory insertion* (OFI).

Interface merging reduces the amount of type information in a program by replacing multiple interfaces by a single one. That way, more classes implement the same interface, and variables and fields whose types are interface types can point to more different types of objects. The object factory insertion transformation replaces object creations by calls to obfuscated factories that can call many different types of constructors to return many different types of objects. In combination with CHF and IM, OFI makes it more difficult to narrow down the exact type of object pointed to by fields, local variables and method parameters at different points in the program without actually executing it. By hiding this information from an attacker, he will not only have a more difficult time understanding the program, but it will also reduce the effectiveness of his tools that rely on precise type information.

To evaluate the protection-wise effectiveness of our obfuscations, as well as the overhead they introduce in transformed applications, we implemented them in a prototype obfuscator, which we used to obfuscate real-world Java applications from the DaCapo benchmark suite. Using software complexity metrics from the domain of software engineering we demonstrate that our obfuscations effectively reduce the understandability of transformed program versions, with program versions becoming less understandable as more aggressive obfuscation settings are used. Additionally, we show that our techniques increase the points-to set sizes of variables, parameters and fields in applications up to twelve times, thereby significantly reducing the effectiveness of static analysis techniques that rely on precise type information.

Like many other obfuscating transformations, ours can also come at a considerable cost in terms of application size and run-time overhead. For most benchmarks, the execution time overhead and the additional memory required to during execution are limited. However, some heavily obfuscated versions of one benchmark were more than eight times slower than the original version of the benchmark. For almost all the application size overhead was very large, with increases of up to 700%.

Experiments showed that most of the execution time overhead was caused by object factory insertion, whereas most of the application size overhead was caused by method merging (MM), an additional technique we developed to reduce the application size overhead of class hierarchy flattening and interface merging. Both techniques involve merging the signatures of methods, be it either to create factory methods that are able to invoke many different constructors, or to create merged methods by combining the parameter type lists of two methods. In both cases the parameter type lists of methods can become so large that they result in significant overheads. For OFI the large number of parameters means that invoking a factory method is many times more expensive than invoking the individual constructors, as many more arguments need to be passed each time an object has to be created. For MM the large parameter type lists often mean that more space is required to store the merged parameter type lists as part of the application's meta-data than the transformation is able to save by merging the methods.

To reduce the overhead of these transformations, we addressed several of their flaws. We made a number of improvements to the method merging algorithm and extended it with a model that enables it to accurately estimate the impact of different potential merge operations on an application's size, so that it can choose the best possible ones, and actually stop merging methods when this is no longer beneficial. Additionally, we modified OFI such that it is able to distribute constructors over multiple factory methods, based on how many times each constructor is invoked, instead of generating a single factory method that invokes all constructors. That way, the algorithm can group frequently invoked constructors in factory methods that require few arguments, to reduce the overall execution time overhead.

An evaluation of our improved techniques shows that they offer comparable levels of protection as their original versions, but at much lower overheads. On average, our improvements reduce application size overhead by over 30%, and execution time overhead by over 40%. Given these large reductions, our improved techniques can now be used not only to offer comparable levels of protection at lower overheads compared to their original versions, but also to offer more protection at similar levels of overhead.

Contents

Chapter 1

Introduction

The business model of many companies that distribute software as (part of) their main products relies on the fact that their software is not tampered with, and that any proprietary data and algorithms embedded in it remain secret. However, trying to achieve this in today's world is proving to be a difficult problem, as reverse engineering and modification of software are well-understood and common practices. With a few targeted Internet searches malicious users can easily obtain a wide array of free reverse engineering tools, including disassemblers [3, 25, 49], debuggers [2, 6], and decompilers [1, 4, 20], as well as detailed instructions on how to use them. Even state-of-the-art commercial reverse engineering tools [31, 41] can easily be obtained through illegal channels. Malicious users can use these tools with the goals of software piracy, software IP theft, data theft, and malware injection.

Fortunately, despite the abundance of attack tools, attackers often still have to invest time and effort to understand an application's code at least partially before being able to modify it, abuse it, or extract valuable information from it. How much time and effort is required greatly depends on the experience of the attacker, the quality of existing tools, his familiarity with those tools, his ability to create his own tools, his familiarity with the libraries used by the application, and the target platform for which it was compiled, as well as the complexity of the code, and whether or not any protection mechanisms are in place. Nowadays, these last two often go hand in hand; software protection mechanisms are put in place to make code artificially more complex, in an effort to ward off potential attackers. In practice, these protection mechanisms are commonly referred to as obfuscations, or obfuscating transformations. Collberg et al. [22] more formally define these transformations as follows.

Definition 1.1. *Obfuscating transformation*

Let $P \xrightarrow{T} P'$ be a transformation of a source program P into a target program P'. $P \xrightarrow{T} P'$ is an obfuscating transformation, if P and P' have the same observable behavior. More precisely, in order for $P \xrightarrow{T} P'$ to be a legal obfuscating transformation the following conditions must hold:

- If P fails to terminate or terminates with an error condition, then P' may or may not terminate.
- Otherwise, P' must terminate and produce the same output as P.

Collberg et al. [22] define observable behavior loosely as behavior experienced by the user. This means that the transformed program may have certain side effects that the original program does not have, as long as these side effects are not experienced by the user.

However, under this definition any behavior preserving transformation is an obfuscating transformation. Ideally, obfuscating transformations should be potent and resilient [22]. That is, they should make it more difficult to understand programs or to extract certain information from them, and it should be difficult to undo these transformations by means of an automatic deobfuscator. Another desirable feature is low cost, which means that transformations should have little impact on the performance and/or size of the resulting applications. Stealth, which is important for transformations that add, e.g. run-time checks to implement integrity verification for tamper-detection, is often less important for obfuscation transformations. In fact, some of the most effective obfuscation transformations are also some of the least stealthy.

In an ideal world, obfuscators transform programs in such a way that an attacker is unable to learn anything from them except for their input-output behavior. However, realistically speaking, such complete obfuscation is impossible in general. Barak et al. [11] proved this by constructing a family of functions that cannot be obfuscated completely. Despite this proof, obfuscations still prove useful in practice. They may not be able to completely stop an attacker, but they may make attacks too complex or too time-consuming to be economically viable. With this goal in mind, many obfuscation techniques have been developed in the past. Collberg et al. [22] divide them into four categories.

Layout transformations target information that is not required for the correct operation of programs [12, 18]. They remove debugging information such as line number information and information about local variables, or replace it by bogus information to cause confusion. Some layout transformations operate on source code programs, in which case they may change the formatting, remove the comments, or scramble the identifiers. Nowadays, they are commonly used to obfuscate JavaScript code. Certain layout transformations can also be applied to programs distributed in bytecode formats, such as Java bytecode or CIL [56].

Control transformations make the control flow of a program appear more complex [22, 23, 43, 52, 53, 59, 62, 75]. These transformations generally operate on the control flow graphs of an application's functions, making them more complex by adding additional nodes and edges, or by splitting or merging them. To obfuscate control flow, researchers have also suggested transformations from the domain of compiler optimizations, including inlining, outlining, and several loop optimizations, such as loop unrolling and loop tiling [7].

Several control transformations rely on *opaque predicates* [23]. These predicates are constructed in such a way that their value is known at obfuscation time (when they are inserted in the program), but for which it is difficult to prove what their value will be during the execution of the program. Commonly used opaque predicates are those that either always evaluate to true, or to false. These so-called one-way predicates can be used to guard complex (nonsensical) code that was inserted by an obfuscator, but that should never be executed.

Data transformations target the data values used and produced by a program [22, 80]. Their goal is to obscure the data values in such a way that they only make sense to the program and not to an attacker. Examples of data transformations include variable splitting and merging, and converting static data into procedural data, such that it is generated at run time [22].

Preventive transformations target tools that are used for reverse engineering, such as decompilers. Their goal is to introduce code that exploits certain weaknesses in these tools to make them generate incorrect source code, or even crash them [12]. Some of these transformations target specific versions of certain tools, which may limit their resilience when other reverse engineering tools are used.

To protect against decompilers and static analysis tools, a specific class of preventive transformations typically implemented by so-called *packers* [45] replace an application by a stub that decompresses and/or decrypts the actual application code at run time. Without access to the specific algorithms used for compression and encryption (and the required decryption keys), the actual program code cannot be uncovered unless at least part of the program is executed.

With the exception of the last category, which specifically targets tools, obfuscation transformations either target an application's control, its data, or its meta-data. To obfuscate binary C and C++ programs it is often sufficient to only use control and data obfuscations. This is because binary programs are typically stripped to remove symbolic information, which means that they contain little meta-data that is easily accessible to an attacker. The meta-data that remains is often implicitly embedded in the program, which may make it difficult to retrieve. For instance, with the exception of those compiled to contain run-time type information (RTTI) [69], C++ programs do not explicitly contain the subtype relations between their classes. In fact, the concept of classes does not even exist anymore in the compiled binaries. Instead, (part of) the subtype information is implicitly embedded when creating a program's virtual function pointer tables, or *vtables* during compilation. Recovering the original class hierarchy from an application's vtables may require some effort.

1.1 Bytecode Obfuscation

For Java bytecode applications, control and data transformations alone do not provide sufficient protection. This is because all of their meta-data is encoded explicitly, such that it can be used by virtual machines to execute the applications and to support features such as bytecode verification, just-in-time (JIT) compilation, reflection, and garbage collection. Even though such features aim at increasing programmer productivity, by not requiring programmers to manually fine-tune their code or concern themselves with garbage collection, they do come at a high cost in terms of information that is made available to a potential attacker. In fact, bytecode applications contain so much meta-information that unobfuscated ones can be decompiled into source code programs that closely resemble the programs before compilation.

To remove some of this information, software protection researchers have proposed layout transformations to obfuscate the identifiers of methods, fields and classes, and the names of packages [12, 18]. However, the meta-data of a bytecode application consists of much more than just identifier names; it also contains type information and information about the application's design. Just as this information helped the original programmers manage the complexity of the program, it can also improve an attacker's understanding of how the program works. This is because during development each class is carefully constructed to model a single logical unit of an application, and the inheritance relations and interactions between the classes are chosen in a meaningful way such that they contribute to the overall understanding of the program. Hence, even though an attacker may not have access to the original identifier names after they have been scrambled, he still has access to a program in which each class represents a well-defined logical unit, and for which the corresponding class file contains a complete description of the class' members, including their signatures, and references to class' superclass and the set of interfaces it implements.

To provide an additional layer of protection, several techniques can be used to also obfuscate a program's design and type information. Sosonkin et al. [66] propose three design obfuscation techniques; *class coalescing*, *class splitting*, and *type hiding*. The first two can be used to merge multiple classes into one, or split one class into several different ones, respectively. The type hiding transformation makes each class implement several interfaces. That way, instances of those classes can be used as instances of many different types throughout the program.

Gone and Stamp [37] combine design pattern detection techniques with class coalescing and class splitting to hide commonly used, and easily recognizable, well-understood software design patterns [36]. Furthermore, the *false factoring* transformation by Collberg et al. [22] takes advantage of the fact that classes are perceived to be related in some way when they share a common ancestor. It makes originally unrelated classes share the same superclasses to make it seem as if they are related.

In the work presented in this dissertation, 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. Furthermore, 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). Interface merging reduces the amount of type information in a program by replacing different interfaces by a single one. That way, more classes implement the same interface, and variables and fields can store more different types of objects. In combination with the object factory insertion transformation, which replaces object creations by calls to obfuscated factories that can return many different types of objects, it becomes more difficult to narrow down the exact types of objects stored in fields, local variables, and method parameters at different points in the program. By hiding this information from an attacker, he will not only have a more difficult time understanding the program, but it will also reduce the effectiveness of his tools that rely on precise type information.

1.2 Motivating Example

We illustrate the issues our transformations tackle by means of an example media player application. The application consists of three parts: the player initializer, support for media files, and support for media streams in those files. Figure 1.1 shows the corresponding class hierarchy subtrees. Figure 1.2 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 [12, 18].

The main method of class Player creates an array of MediaFile objects to be played (line 10). It then queries them for their media streams (line 12), which are initialized by accessing the file with the readFile method. Figure 1.2 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 (lines 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 (lines 14 & 16). The play methods essentially output the raw bytes of the media streams to a specific output device. Those bytes are obtained, decrypted (lines 33–34) and decoded (line 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

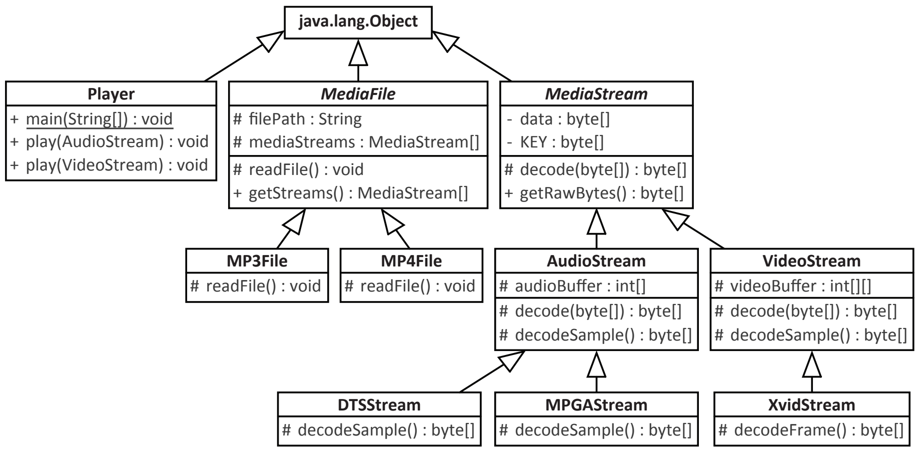


Figure 1.1: Standard UML representation of the class hierarchy of a simple DRM media player.

by subclasses of MediaStream. The decryption process, by contrast, 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 superclass. To further improve the quality of the code, we could have also factored out casts and run-time type checks. However, we chose not to 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. For instance, all media streams are decrypted with MediaStream.getRawBytes(). Hence, 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 line 9 allows a type inference tool [13] to narrow the type of the player variable to Player. The instanceof checks and casts on lines 13–16 also restrict the possible types of objects to which the variable ms can point. This abundance of type information is important for an attacker because it

```
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[] data = new byte[...];
    inputStream.read(data);
    AudioStream as = new MPGAStream(data);
    mediaStreams = new MediaStream[]{as};
    return;
  }
}
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);
}
```

Figure 1.2: Partial implementations of the original Player, MediaStream and MP3File classes.

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 [67].

These issues can be solved by rewriting the well-structured hierarchy into the unstructured collection of Figure 1.3. 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

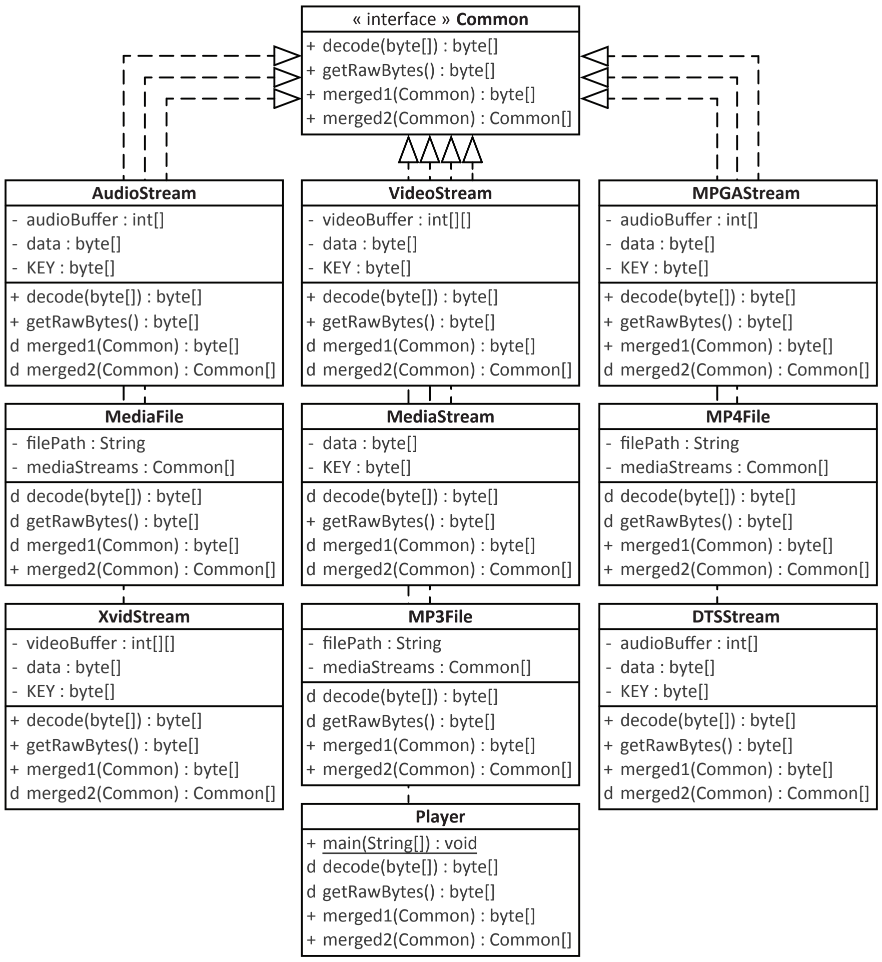

« interface » Common
+ decode(byte[]) : byte[]
+ getRawBytes() : byte[]
+ merged1(Common) : byte[]
+ merged2(Common) : Common[]
AudioStream
- audioBuffer : int[]
- data : byte[]
- KEY : byte[]
+ decode(byte[]) : byte[]
+ getRawBytes() : byte[]
d merged1(Common) : byte[]
d merged2(Common) : Common[]
VideoStream
- 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[]
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[]
XvidStream
- videoBuffer : 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[]
DTSStream
- audioBuffer : int[]
- data : byte[]
- KEY : byte[]
+ decode(byte[]) : byte[]
+ getRawBytes() : byte[]
+ merged1(Common) : byte[]
d merged2(Common) : Common[]
Player
+ main(String[]) : void
d decode(byte[]) : byte[]
d getRawBytes() : byte[]
+ merged1(Common) : byte[]
+ merged2(Common) : Common[]

Figure 1.3: Type-obfuscated class hierarchy of the media player.

between related classes, so one can no longer attack many classes at once by patching their common superclass.

Code analysis has also become harder. Figure 1.4 displays much less type information than the original code. All declarations declare type Common, all invoked methods are implemented by all classes, and all casts have been removed. An obfuscated, typeless isInstance method replaces instanceof, and factories returning instances of type 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 [40], points-to

```
public class Player implements Common {
  public byte[] merged1(Common as) {
    /* send as.getRawBytes() to audio device */
  }
  public Common[] merged2(Common vs) {
    /* send vs.getRawBytes() to video device */
  }
  public static void main(String[] args) {
    Common player = CommonFactory.create(...);
    Common[] mediaFiles = ...;
    for (Common mf : mediaFiles)
      for (Common ms : mf.getStreams())
        if (myCheck.isInst(0, 1, ms.getClass()))
          player.merged1(ms);
        else if (myCheck.isInst(1, 1, ms.getClass()))
          player.merged2(ms);
  }
}
public class MP3File implements Common {
  public byte[] merged1() {
    InputStream inputStream = ...;
    byte[] data = new byte[...];
    inputStream.read(data);
    Common as = CommonFactory.create(...);
    mediaStreams = new Common[]{as};
    return data;
  }
}
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 byte[] decode(byte[] data){ ... }
}
```

Figure 1.4: Type-obfuscated versions of the Player, MediaStream and MP3File classes. Transformed code is shown in red.

analyses [67] and type inference [13] 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 (indicated by the letter 'd' instead of their visibility modifier in Figure 1.3) that can be filled with arbitrary code to complicate static analysis even further.

In the next chapters we discuss the stepwise code obfuscation. Class hierarchy flattening (Chapter 2) first replaces the type hierarchy by a flat collection of classes. In doing so, it introduces a single interface for each subtree of the original class hierarchy. However, despite the use of these interface types, the flattened program may still encode a considerable amount of type information. We therefore use interface merging (Chapter 3) to merge separate interfaces into a common one. As a result, method signatures feature less diverse types and many casts can be removed. This leads to a reduction in type information and enables object factory insertion (Chapter 4) to further remove type information from object allocation sites for optimal protection. Furthermore, to reduce the overhead of our transformations and improve their practical usefulness, we present several improvements to them in Chapter 6.

1.3 Contributions

The major contributions of this dissertation are the following.

- *Better protection.* We present class hierarchy flattening, interface merging, and object factory insertion, three complementary obfuscation techniques that obfuscate much more type information than several existing obfuscations. Our transformations not only make code more difficult to understand by a human attacker, but they also make automatic analysis tools less effective. In some cases they even improve the applicability of other transformations.

- *Cheaper protection.* We present effective methods and heuristics to limit the overhead of our transformations and to trade it off for the level of protection. On some levels, our transformations are not only more cost-effective than some existing obfuscations, they can also reduce the cost of some of these transformations.

- *Automatic protection.* We present a tool flow that is able to apply our obfuscations fully automatically on complex, real-world applications that heavily depend on reflection and custom class loaders. However, to operate correctly our tool flow does require that the entire application is available at obfuscation time.

- *Measurable protection.* Our evaluation includes metrics related to human code understanding as well as to automated static analysis tools that help attackers reverse-engineer bytecode. To the best of our knowledge, we are the first to evaluate and report obfuscations of this complexity on large, real-world applications.

In Chapter 2 we explain class hierarchy flattening in more detail. In doing so, we pay special attention to previously unpublished transformation steps, such as how static initializers are handled, how array types are updated, and how exception classes can be transformed. The work in this chapter has led to the following conference publication.

A Novel Obfuscation: Class Hierarchy Flattening
Christophe Foket, Bjorn De Sutter, Bart Coppens, and Koen De Bosschere
In *International Symposium on Foundations & Practice of Security, 2012 [32].*

Chapters 3 and 4 report on interface merging and object factory insertion, respectively. In Chapter 4, we also explain in more detail our version of the type inference algorithm by Gagnon et al. [35], which we use to create more effective object factories. The work in Chapters 3 and 4, combined with the work in Chapter 2 and the evaluation of our techniques in Chapter 5, has led to the following journal publication.

Pushing Java Type Obfuscation to the Limit
Christophe Foket, Bjorn De Sutter, and Koen De Bosschere
In *IEEE Transactions on Dependable and Secure Computing, 2014 [33].*

In Chapter 6, we present and evaluate several improvements for the transformations described in Chapters 2 to 4 to reduce the code size and execution time overhead of the transformed applications. As part of the work in this chapter, we present a model for tracking changes in the size of an application during transformation, as well as a way of managing the overhead of object factory insertion. The work in Chapter 6 has led to a journal article that is currently under submission.

Cost-effective Java Type Obfuscation
Christophe Foket, Bjorn De Sutter, and Koen De Bosschere
Submitted to *IEEE Transactions on Dependable and Secure Computing*, July 2015.

For our obfuscation tool flow, we rely on the following open-source tools, to which we have made several contributions.

Soot is a program analysis and transformation framework with support for Java source code, Java bytecode, and Android bytecode [74]. It features several static analysis techniques, including call-graph construction techniques, points-to analyses, and liveness analysis. It also offers support for template-driven intra-procedural and inter-procedural data-flow analyses [15]. Over the years, several tools have been built around Soot, of which the Java ByteCode Obfuscator (JBCO) [12] and the Design Obfuscator for Java (DOJ) [66] are two examples.

Unlike many other bytecode transformation frameworks, Soot does not require users to work close to or at the bytecode level [17, 27]. Instead, users can choose from several intermediate representations at different abstraction levels. Unless otherwise noted, all transformations presented in this dissertation operate on Jimple code, a three-address intermediate representation that is both high level and easy to transform.

TamiFlex is a collection of tools created to facilitate the analysis and transformation of applications in the context of reflection and custom class loaders [16]. In this work we primarily rely on two of the tools: the Play-out Agent and Booster. The TamiFlex Play-out Agent is a dynamic instrumentation tool that outputs all classes that are loaded during the execution of an application, and a list of all reflective operations performed during the execution. Booster is a static tool that uses the list of reflective operations output by the Play-out Agent to inline a program's reflective operations, replacing them by direct operations on the involved classes, methods, and fields, such that static analysis tools that cannot model reflective operations can still analyze the program.

WALA is a static analysis framework, primarily oriented at analyzing Java bytecode [30]. It features highly configurable implementations of state-of-the-art pointer analyses and call graph construction algorithms. We mainly use WALA as an attack tool to evaluate how well our transformations succeed in confusing static analyses, in an effort to measure our transformation's effectiveness against automated attacks.

We give a detailed overview of our contributions to these tools, as well as several other tools that we developed to evaluate our transformations when we discuss our experimental setup in Chapter 7.

Chapter 2

Class Hierarchy Flattening

In this chapter we discuss class hierarchy flattening. The goal of this transformation is to remove as many subtype relations from a class hierarchy as possible. For example, in our media player example introduced in the previous chapter, MP3File and MP4File should no longer inherit from MediaFile. In practice not all subtype relations can be removed, however. 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 that can be flattened are selected from an application's class hierarchy. In five following steps, all necessary transformations are performed to flatten the selected subtrees without changing the behavior of the application.

2.1 Subtree Selection

Assume that an application consists of a set of *application types* (i.e. classes and interfaces) $\mathbb{A}$ that use or extend classes and interfaces from a self-contained set of *library types* $\mathbb{L}$ that includes java.lang.Object. Types in $\mathbb{L}$ are never considered for transformation. In practice, $\mathbb{L}$ usually corresponds to the standard library, while $\mathbb{A}$ contains all classes and interfaces that make up the actual application. Let $t_s, t^s : (\mathbb{A} \cup \mathbb{L}) \mapsto \mathcal{P}(\mathbb{A} \cup \mathbb{L})$ be the functions that map a type x to the sets $t_s(x)$ and $t^s(x)$ of all x's (transitive) subtypes and supertypes, respectively, 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 in

$t_s(l)$ with l a library class needs to stay a subclass of l. This is similar to limitations imposed on other refactorings. Those limitations have been formalized in literature [73], so we do not repeat them here. Furthermore, CHF is not applicable to a specific subset of classes $\mathbb{X} \subset \mathbb{A}$ because changing those classes' position in the hierarchy could alter 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. For instance, our prototype tool considers classes of which multiple different or identical definitions exist on the class path as non-transformable, as well as classes that are subtypes of java.lang.Throwable. The latter are usually exception classes. Implementing tool support for these classes would require a large engineering effort and not buy much in terms of obfuscation. That aside, ensuring that these classes can be transformed is conceptually not very difficult; we provide an algorithm for doing this in Section 2.8. Note, however, that this algorithm is currently not implemented in our prototype tool. Instead, our tool simply adds classes that (indirectly) extend java.lang.Throwable to $\mathbb{X}$ as non-transformable. All classes in $\mathbb{X}$ face similar limitations as those in $\mathbb{L}$.

During subtree selection the classes in $\mathbb{A}$ are partitioned into a set $\mathbb{T}$ of transformable classes and a set $\mathbb{X}$ of non-transformable classes. $\mathbb{T}$ is then further partitioned into disjoint subtrees T_i according to the following four rules:

align center

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

These rules 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. Figure 2.1 depicts the selection of four subtrees in a hierarchy with an external library class L and a non-transformable

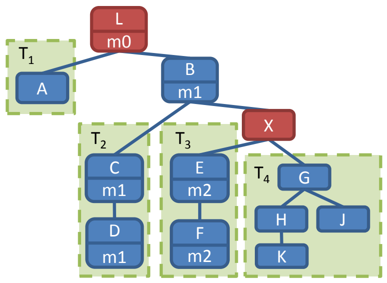


Figure 2.1: Selected subtrees in a class hierarchy.

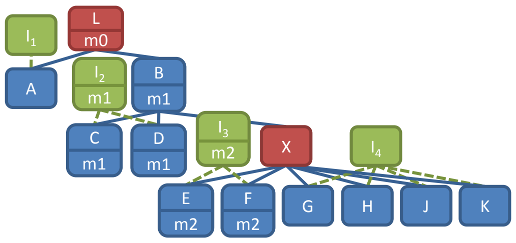


Figure 2.2: Flattened class hierarchy subtrees.

application class X. In the media player of Figure 1.1, the three subtrees of java.lang.Object are selected.

In the following five sections we explain how each subtree is transformed into a flat set of classes that become siblings of the tree's root. For each flat set of classes, our algorithm inserts an interface that these classes implement. Figure 2.2 shows the result with four new interfaces for the original class hierarchy of Figure 2.1.

2.2 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[1]. 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

[1]For brevity, we did not perform this step in our running example.

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.

Finally, we rewrite the static initializers of the subtree classes to preserve the order in which classes are initialized. In Java, a class is initialized automatically by the Java Virtual Machine the first time one of its methods is invoked, one of its fields is accessed, or one of its subclasses is initialized[2]. In the latter case, the static initializer of the class is invoked automatically before the static initializer of its subclass. To ensure that the initialization order of the classes is preserved after flattening, we rewrite the program such that the static initializer of each class is executed before the static initializer of its subclasses in a way that does not depend on the subtype relations between the classes. Unfortunately, the Java Virtual Machine Specification does not allow programs to call static initializers directly. The static initializer of a class can only be invoked by the virtual machine. To preserve the classes' initialization order we therefore rewrite the static initializers of subtree classes such that each class' static initializer performs an operation on that class' superclass that triggers the initialization of the superclass. We do this by computing for each class $c \in \mathbb{T}$ the closest ancestor d in the same subtree as c that has a static initializer. If d exists, we proceed as follows:

1. Let f be a random public static field of d. If d does not contain any public static fields, create one first.
2. Let m be the static initializer method of c. If c does not contain such a method, create it first.
3. Prepend the code in m with code that reads field f.

To cause the Java Virtual Machine to invoke the static initializer of d we can also insert code that writes to one of d's fields, calls one of its

[2]These are the most common events that trigger class initialization. For a complete overview, we refer to the Java Virtual Machine Specification [48].

methods, or invokes certain reflective methods on it. We chose to add code that reads one of d's fields because it is easy to implement, and because it is guaranteed not to have side-effects.

2.3 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 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, respectively.

2.4 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[3], 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. Figure 2.3 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.

Like the operation described in Section 2.2, this abstraction operation sometimes requires fields or methods to be renamed, or additional distinguishing parameters to be added to constructors to avoid cases in which a class defines multiple members with the same signature. Consider, for instance, the two play methods in Figure 1.2, which have different parameter types. After changing their parameter types to Common their

[3]In Section 2.7 we explain in more detail how array creation expressions are updated.

```
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);
  }
}
```

Figure 2.3: Intermediate obfuscated Player class.

signatures become identical. To differentiate them, one of them was renamed to play1, as shown in Figure 2.3.

As for cast operations, many of them can be omitted because they have become superfluous after interfaces are used to replace 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 [42, 61]. This treatment of casts is similar to that in other code refactoring techniques that change type hierarchies [73].

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 Figure 2.1, consider a method void m0(L x) of library class L. In the original program, the following code sequence is valid:

$$\mathsf{L\ o = ...;\ A\ a = ...;\ o.m0(a);}.$$

After rewriting the declarations, we have to insert a cast:

$$\mathsf{L\ o = ...;\ I1\ a = ...;\ o.m0((L)a);}.$$

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.

2.5 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 for each direct subclass d of c, we first make d implement the same interfaces as c to preserve assignment-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.

2.6 Converting instanceof

The behavior of run-time type checks inserted by the programmer or during subtree type abstraction depends on the structure of the class hierarchy. Before flattening the subtrees of Figure 1.1, the expression ms instanceof AudioStream on line 13 of Figure 1.2 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 2.1 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. Additional rows can also be added for dummy instanceof operations injected into the dummy methods created in previous steps.

	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 2.1: instanceof lookup table

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 [54], 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 2.1 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 [19] to prevent static analysis of the table and involved code.

Each expression o_i instanceof A_j in the program is then replaced by a call myCheck.isInst($r_{i,j}$, n, o_i.getClass()) where $r_{i,j}$ is the (possibly encrypted or hashed) row index of the lookup table entry that corresponds to the given instanceof expression, and n is the number of dimensions of A_j. The latter is required to handle instanceof expressions involving array types. Consider, for example, the expression o_i instanceof A[][]. For this expression to evaluate to true two conditions must be met. First, the number of dimensions of o_i's type must match the number of dimensions of A[][], which is two. Second, the base type b of o_i's type (i.e., the dimensionless version of this type) must be an element of t_s(A). Or, in other words, objects of type b must be instances of type A. The first condition can be verified by matching the number of dimensions of o_i.getClass() against n. For the second we can use the lookup table.

Lines 13 and 15 of Figure 2.4 show the results for the media player.

2.7 Updating Array Creation Expressions

As part of the subtree type abstraction process described above, we need to correctly update the types used in array creation expressions. Unfortunately, this is not as straightforward as updating the types of local

```
public class Player implements Common1 {
  public void play(Common3 as) {
    /* send as.getRawBytes() to audio device */
  }
  public void play1(Common3 vs) {
    /* send vs.getRawBytes() to video device */
  }
  public static void main(String[] args) {
    Common1 player = new Player();
    Common2[] mediaFiles = ...;
    for (Common2 mf : mediaFiles)
      for (Common3 ms : mf.getStreams())
        if (myCheck.isInst(0, 1, ms.getClass()))
          player.play(ms);
        else if (myCheck.isInst(1, 1, ms.getClass()))
          player.play1(ms);
  }
}
public class MP3File implements Common2 {
  public void readFile() {
    InputStream inputStream = ...;
    byte[] data = new byte[...];
    inputStream.read(data);
    Common3 as = new MPGAStream(data);
    mediaStreams = new Common3[]{as};
    return;
  }
}
public class MediaStream implements Common3 {
  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 byte[] decode(byte[] data){ ... }
}
```

Figure 2.4: Partial implementations of the Player, MediaStream and MP3File classes after class hierarchy flattening. Changes compared to the original code are shown in red.

variables and method parameters. Consider, for instance, the code fragment in Figure 2.5(a), taking into account the class hierarchy shown in Figure 2.1. Assuming for a moment that CHF does not update the types used in array creation expressions, this code fragment would be transformed into the incorrect one shown in Figure 2.5(b). The flattened class hierarchy corresponding to this code fragment is shown in Figure 2.2. Because classes H and K are no longer subclasses of G and H, respectively, lines 4 and 5 of the code fragment will result in ArrayStoreExceptions.

```
1 B[][][] b = new B[1][][];
2 G[][] g = new G[1][];
3 H[] h = new H[1];
4 h[0] = new K();
5 g[0] = h;
6 b[0] = g;
```

(a)

```
1 B[][][] b = new B[1][][];
2 I4[][] g = new G[1][];
3 I4[] h = new H[1];
4 h[0] = new K();
5 g[0] = h;
6 b[0] = (B[][])g;
```

(b)

```
1 B[][][] b = new B[1][][];
2 I4[][] g = new K[1][];
3 I4[] h = new K[1];
4 h[0] = new K();
5 g[0] = h;
6 b[0] = (B[][])g;
```

(c)

Figure 2.5: Updating types used in array creation expressions. (a) Original code. (b) Code after flattening with original types. (c) Code after flattening with updated types. In (b) and (c) a cast was added to ensure type-correctness.

To ensure that the assignment on line 4 succeeds, we could replace new H[1] by new I4[1] on line 3. After doing this, new G[1][] should be replaced by new I4[1][] on line 2 to ensure that the assignment on line 5 is valid. Unfortunately, doing so will result in a ClassCastException on line 6. Replacing new H[1] by new I4[1] is hence not valid, since the behavior of the program is not preserved. Alternatively, we could also replace new H[1] by new K[1]. Updating new G[1][] accordingly results in a valid type assignment, as shown in Figure 2.5(c). Note, however, that this code is only valid because h only stores elements of type K. If, for example, h also has to store elements of type H, the code would not be valid.

The above example illustrates only some of the ways in which a program's behavior can be altered as a result of incorrectly updating the types of its arrays. In practice, updating those types incorrectly will not always result in ClassCastExceptions and ArrayStoreExceptions, which may cause the program to terminate with an error message. It is also possible that such exceptions are masked, or that the result of instanceof expressions changes. In those cases the effects may be more subtle. Nonetheless, the program is then transformed incorrectly.

In summary, to ensure that the behavior of the program is preserved, any algorithm that updates the types used in array creation expressions should ensure that the following conditions hold for each array a.

C.1 All store operations involving a that succeed/fail before transformation must also succeed/fail after transformation.

C.2 All type checking expressions that succeed/fail for a before transformation must also succeed/fail after transformation.

Based on these conditions, one could build an algorithm that models the relationships between the arrays, their elements, and the type checks using a system of type constraints [71]. Such a system would, for instance, express line 4 of Figure 2.5(a) as a constraint that enforces that it should always be possible to store the array created on line 3 in the array created on line 2, regardless of their types. Expressing lines 5 and 6 as similar constraints would then result in a system of constraints that can be solved to obtain the new type for each array.

However, even though an algorithm that builds a constraint system will generally yield the best results, implementing such an algorithm also requires a rather large engineering effort. This effort is especially large when taking into account that in practice only 5% of the classes in an application are used in array creation expressions[4]. Because we did not find this approach worth the extra engineering effort, we instead developed an algorithm that computes the new array types using a few simple rules. Despite being simple and easy to implement, we found it performs really well in practice.

To keep our algorithm simple we make it operate on base types rather than array types. For each base type b it computes a new base type b' such that all expressions that create arrays with base type b can be replaced by ones that create arrays with base type b'. The benefit of this approach is that each array creation expression can simply be represented by the base type of the arrays it creates. The expressions do not need to be taken into account individually, which means our algorithm does not require an allocation-site-sensitive points-to analysis. To further simplify our algorithm, we make it express conditions C.1, and C.2 in terms of sets of types each replacement type should be a subtype/supertype of. In doing so, the new types can be computed directly from those sets, without the need for an iterative constraint solver.

To compute the new array base types our algorithm operates in three steps that are discussed in more detail in the next sections. First, the algorithm computes for each array base type the sets of types that constrain its replacement type. Next, it filters out the base types for which it cannot prove that it can replace them without altering the behavior of the program. Finally, for the remaining types it determines the replacement type from the sets constructed in the first step.

Before we go into the details, it is worth noting that our algorithm may not always find a suitable replacement type for a given set of array

[4]Computed over the 12 applications from the DaCapo benchmark suite that we worked with.

creation expressions. In such cases, it will terminate early after discovering that transforming certain classes may violate one or more of the above conditions. The recommended strategy is then to mark those classes as non-transformable and restart the algorithm. However, when additional classes are marked as non-transformable, some of the subtrees will no longer be valid, which means the subtree selection process and all subsequent steps must be repeated. To avoid this, we propose to include the algorithm as part of the subtree selection process. That way, the algorithm can be invoked iteratively, and classes can be marked as non-transformable until it terminates successfully. At that point, the results of the algorithm can be saved and used to update the types of array creation expressions during the subtree type abstraction step.

2.7.1 Computing the Type Sets

Let $\mathbb{B}$ be the subset of $\mathbb{T}$ (defined in Section 2.1) for which each type occurs as the base type of an array creation expression. Furthermore, for each type $t \in \mathbb{A} \cup \mathbb{L}$, we define the following sets.

- $e(t)$ is the set of base types of all elements that the program tries to store in arrays with base type t. For convenience, we define the base type of a non-array type n to be the type n itself.
- $h(t)$ is the set of base types of all variables and array instances in which the program tries to store array instances with base type t.
- $c(t)$ is the set of base types of all array instances a for which there exists (i) a cast expression that casts a to a type with base type b, or (ii) an instanceof expression that checks the type of a against a type with base type b.

To constrain the replacement types as little as possible, $e(t)$, $h(t)$, and $c(t)$ are best computed using precise points-to information about the program, rather than imprecise points-to information obtained using class hierarchy analysis [29] or rapid type analysis [8]. A type u should hence only be added to $e(t)$ if the program contains a statement $\mathsf{a[i]} = \mathsf{x}$ for which a can point to an array with base type t and x can point to an instance with base type u. Similar restrictions apply to $h(t)$ and $c(t)$.

For the code in Figure 2.5(a) $e(\mathsf{B}) = \{\mathsf{G}\}$, $e(\mathsf{G}) = \{\mathsf{H}\}$, and $e(\mathsf{H}) = \{\mathsf{K}\}$. Additionally, $h(\mathsf{B}) = \{\mathsf{B}\}$, $h(\mathsf{G}) = \{\mathsf{G}, \mathsf{B}\}$, and $h(\mathsf{H}) = \{\mathsf{H}, \mathsf{G}\}$. Finally, $c(t)$ is empty for all types $t \in \{\mathsf{B}, \mathsf{G}, \mathsf{H}\}$.

Algorithm 2.1: Computing $\overline{e}(b)$, $\overline{h}(b)$, and $\overline{c}(b)$.

```
foreach $b \in \mathbb{B}$ do
    $\overline{e}(b) \leftarrow e(b)$
    $\overline{h}(b) \leftarrow h(b)$
    $\overline{c}(b) \leftarrow c(b)$
repeat
    foreach $b \in \mathbb{B}$ do
        foreach $o \in \overline{e}(b)$ do
            $\overline{e}(b) \leftarrow \overline{e}(b) \cup \overline{e}(o)$
        foreach $o \in \overline{h}(b)$ do
            $\overline{h}(b) \leftarrow \overline{h}(b) \cup \overline{h}(o)$
            $\overline{c}(b) \leftarrow \overline{c}(b) \cup \overline{c}(o)$
until $\overline{e}(b)$, $\overline{h}(b)$, and $\overline{c}(b)$ did not change during this iteration;
```

For each type $b \in \mathbb{B}$, our algorithm computes $\overline{e}(b)$, $\overline{h}(b)$, and $\overline{c}(b)$, the transitive closures of $e(b)$, $h(b)$, and $c(b)$, respectively, as shown in Algorithm 2.1. The use of transitive closures is our algorithm's way of modeling how arrays of different types are used in relation to each other. It enables the algorithm to take into account type information about (i) all instances that can be (in)direct elements of an array a with a certain base type, (ii) all the arrays of which a can be an (in)direct element, and (iii) all type checks and casts involving those arrays.

2.7.2 Filtering Out Problematic Cases

To simplify the computation of the replacement types, our algorithm filters out cases where updating the base type of arrays could affect the result of instanceof expressions or mask potential ArrayStoreExceptions or ClassCastExceptions. It does this by marking certain classes as non-transformable if it cannot prove that all stores and type checking operations involving arrays of those classes will always succeed. This happens as follows.

1. Create a new empty set $\mathbb{Y}$.
2. For each type $b \in \mathbb{B}$, add b to $\mathbb{Y}$ if
    (a) $\overline{e}(b) \not\subseteq t_s(b)$, or

(b) $\overline{h}(b) \not\subseteq t^s(b)$, or

(c) $\overline{c}(b) \not\subseteq t^s(b)$.

3. If $\mathbb{Y}$ is not empty, mark all types in $\mathbb{Y}$ as non-transformable and terminate early. In this case the algorithm should be restarted.

If the algorithm does not terminate early, the result of this step is that the replacement type b' for a given base type b can be determined by only taking into account the types of which b' should be a subtype/supertype. The algorithm does not need to take into account the set of types of which b' cannot be subtype/supertype, which makes determining b' easier.

Note that it is sufficient to perform the checks in steps 2(a) and 2(b) using the sets $t_s(b)$, $t^s(b)$, $\overline{e}(b)$, and $\overline{h}(b)$, all of which contain base types. We do not need the original array types to also check their dimensions. The fact that the bytecode of the program under transformation is valid already guarantees that for each assignment $\mathsf{a} = \mathsf{x}$ and each array store operation $\mathsf{a[i]} = \mathsf{x}$ the dimensions of a and x are compatible.

For type checking expressions we do not have this guarantee. As a result, our algorithm may not always filter out all cases for which instanceof expressions or casts may fail. However, in practice this is not a problem since it only happens for type checks and casts that always fail for a particular array, regardless of its type. Consider, for instance the expression (X[][])a, which casts an n-dimensional array a with base type b. The only way this cast can fail when $b \in t_s(\mathsf{X})$ is if the dimensions of the array types do not match, i.e., if n is not equal to 2. However, this also means that replacing b by any other type will always cause the cast to fail, because the dimensions of the array types will never match. The behavior of this cast can hence not be affected by the transformation, which means it is not necessary to mark b as a non-transformable class.

2.7.3 Computing the Replacement Types

For each type $b \in \mathbb{B}$, let the T be subtree to which b belongs. To compute the type b' by which b should be replaced in array creation expressions, we use the following prioritized set of rules.

R.1 i, the subtree interface of T if

(a) $\overline{c}(b) \subseteq \{\mathsf{java.lang.Object}\}$, and

(b) $\overline{h}(b) \subseteq \{\mathsf{java.lang.Object}\} \cup T$.

b	$\overline{e}(b)$	$\overline{h}(b)$	$\overline{c}(b)$	b'
G	$\{\mathsf{H},\mathsf{K}\}$	$\{\mathsf{B},\mathsf{G}\}$	$\emptyset$	S
H	$\{\mathsf{K}\}$	$\{\mathsf{B},\mathsf{G},\mathsf{H}\}$	$\emptyset$	K

Table 2.2: $\overline{e}(b)$, $\overline{h}(b)$, $\overline{c}(b)$, and new array base types for the code in Figure 2.5(a).

R.2 b, if $\overline{e}(b) = \emptyset$.

R.3 x, if $\overline{e}(b) = \{x\}$.

R.4 s, a new subtree superclass of T if

(a) $\forall\, u \in T \,.\, \overline{c}(b) \subseteq t^s(u)$, and

(b) $\forall\, u \in T \,.\, (\overline{h}(b) \setminus T) \subseteq t^s(u)$.

If rule 4 applies, a new abstract class[5] s should be created that is inserted directly above the root of the subtree T, such that it is a common superclass of all classes in T. Furthermore, s should also implement i, the subtree interface of T, as well as all interfaces in $\overline{c}(b)$ and $\overline{h}(b) \setminus T$.

If there are multiple possible replacement types for the same type b, the one corresponding to the first matching rule is chosen. That way rule 1 has priority over rule 4, which also holds whenever rule 1 holds. As a result fewer subtree superclasses are created. This is desirable because each subtree superclass gives away some type information.

If none of the above rules apply for a type, that type is marked as non-transformable, and the algorithm terminates early. In that case new subtrees should be selected, and the algorithm should be invoked again, as discussed above.

2.7.4 Example

To illustrate how our algorithm works, we demonstrate how it computes new types for the array creation expressions shown in Figure 2.5(a). Table 2.2 shows the sets $\overline{e}(b)$, $\overline{h}(b)$ and $\overline{c}(b)$ for each type $b \in \mathbb{B} = \{\mathsf{G},\mathsf{K}\}$. Note that as a result of the transitive closure, $\overline{e}(\mathsf{G})$ also contains K, which means this type is also taken into account when computing the replacement type for G.

[5]We use an abstract class such that it does not have to implement any of the methods declared in its interfaces.

By applying the rules formulated in step 3 of the algorithm, we obtain the replacement types shown in the last column of Table 2.2. For G rule 4 applies, so a new subtree supertype S is created. For H rule 3 applies, so the corresponding replacement type is K. Note that even though this solution differs from the one shown in Figure 2.5(c), it is still valid. It is, however, less optimal because a new subtree supertype S needs to be created. Our algorithm did not find the optimal solution because it computes the replacement type for each base type individually. As a result, the fact that K is a suitable replacement type for H is not taken into account when determining the replacement type for G. This means that $\overline{e}(\mathsf{G})$ cannot be simplified to $\{\mathsf{K}\}$, and that rule 4 applies instead of rule 3. However, in practice this is not a problem, since rule 4 only applies in around 4% of the cases, on average.

2.7.5 Correctness

To prove that the above algorithm is correct, we must show that each array is assigned a new base type in such a way that program behavior is preserved. Because our algorithm filters out type checks and array stores that may fail, it is sufficient to prove Theorem 2.1 and Theorem 2.2, which are less strict versions of conditions C.1 and C.2. The proof of Theorem 2.1 is based on the following lemmas, for which the proofs can be found in Appendix A.

Lemma 2.1. $\forall\, b \in \mathbb{B}\ .\ \overline{e}(b) \subseteq t_s(b')$ *after class hierarchy flattening.*

Lemma 2.2. $\forall\, b \in \mathbb{B}\ .\ (\overline{h}(b) \setminus \mathbb{T}) \subseteq t^s(b')$ *after class hierarchy flattening.*

Lemma 2.3. *For each array with base type $b \in \mathbb{B}$ in which an array with base type $c \in \mathbb{B}$ is stored, it holds that $c' \in t_s(b')$ after class hierarchy flattening.*

Theorem 2.1. *All store operations involving arrays with base type $b \in \mathbb{B}$ succeed after class hierarchy flattening.*

Proof. For this theorem to hold, the following two statements must hold.

1. Each array with base type $b \in \mathbb{B}$ in the untransformed program can store the same elements after flattening. This statement holds because of Lemma 2.1.

2. Each array with base type $b \in \mathbb{B}$ in the untransformed program can be stored in the same variables v and arrays a after flattening. To prove that this statement holds we make a distinction based on whether or not the base types of the variables v and arrays a belong to $\mathbb{T}$ in the original program.

   (a) If the base types of v and a are not in $\mathbb{T}$ in the original program, the statement holds because of Lemma 2.2.

   (b) If t is the base type of v in the original program, with $t \in \mathbb{T}$, let i be the interface of the subtree to which t belongs. In this case the condition holds because the base type of v is replaced by i during subtree type abstraction, and our algorithm only computes new base types that are elements of $t^s(i)$ after transformation.

   (c) If t is the base type of a in the original program, with $t \in \mathbb{T}$, and hence $t \in \mathbb{B}$ by construction, the statement holds because of Lemma 2.3.

Since both statements hold, the theorem also holds. □

Theorem 2.2. *The behavior of all type checking expressions involving arrays with base types $b \in \mathbb{B}$ is unaffected by our algorithm.*

Proof. First, the theorem holds for rule 1 because condition R.1(a) ensures that there are no type checks of arrays with base type b against any type with a base type other than java.lang.Object. This is because the replacement type i is an interface, whose only supertype is java.lang.Object.

Second, the theorem also holds for rule 4 because of condition R.4(a), which states that each type in $\overline{c}(b)$ should be a supertype of all classes in the subtree that contains b, and the fact that the subtree superclass s is a subtype of all types in $\overline{c}(b)$. As a result, all subtype relations between the types in b's subtree and the types in $\overline{c}(b)$ are preserved.

Finally, the theorem holds for rules 2 and 3. In both cases b is replaced by a type b' that has at least the same supertypes as b in the original program. As a result, any checks of b' against types outside b's subtree will be unaffected. Additionally, any checks of b' against types inside the subtree are handled using our custom instanceof implementation described in Section 2.6. Note that even though b may be replaced by one of its subtypes, there is no risk of type checks suddenly succeeding where they failed in the original program, since our algorithm filters out

```
public void m(){
  stmt0
  try{
    stmt1
    try {
      stmt2
    }
    catch(ExceptionA ex0){
      handler0
    }
    catch(ExceptionB ex1){
      handler1
    }
    stmt3
  }
  catch(ExceptionC ex2){
    handler2
  }
  stmt4
  return
}
```

(a)

```
 1 public void m(){
 2   stmt0
 3   stmt1
 4   stmt2
 5   goto 9
 6   handler0
 7   goto 9
 8   handler1
 9   stmt3
10   goto 12
11   handler2
12   stmt4
13   return
14 }
```

Exception table:

from	to	target	type
4	5	6	Class ExceptionA
4	5	8	Class ExceptionB
3	10	11	Class ExceptionC

(b)

Figure 2.6: Example program with exception handlers. (a) Pseudo-Java code. (b) Pseudo-bytecode.

such cases. Also note that any casts (t[][])o, with $t \in \mathbb{T}$ do not require special treatment since they are converted to (i[][])o, with i the interface type corresponding to the subtree to which t belongs, and guarded by an expression o instanceof t[][], as discussed in Section 2.4. □

2.8 Flattening Throwable Classes

Direct and indirect subclasses of class java.lang.Throwable differ from other classes in that the virtual machine uses their subtype relations to determine which exception handlers to invoke. We illustrate how this happens using the method in Figure 2.6(a). This method's code contains exception handlers for three different types of exceptions: ExceptionA, ExceptionB, and ExceptionC, all of which are (in)direct subclasses of java.lang.Throwable. When an exception is thrown, the virtual machine decides which of the handlers to invoke based on the type of the exception, and where it was thrown. Assume, for instance, that an exception of type X is thrown while executing stmt2. In that case, the virtual machine will traverse all exception handlers for stmt2, and invoke the first one that can handle exceptions of type X. Exception handlers are traversed in a specific order: In source code terms, traversal starts at the most deeply nested try block that contains stmt2. For that block, all associated catch blocks are traversed in declaration order. If there is a catch block for which the thrown exception is an instance of the declared exception type, control is transferred to that block. If no such block can be found, traversal continues at a higher nesting level. If no handler can be found at the highest level, the exception is passed to the calling method, or the program terminates if there is no calling method. If we assume that in our case X is a subtype of ExceptionB, control will be transferred to handler2. That is, of course, assuming that X is not also a subtype of ExceptionA, in which case control would be transferred to handler1.

Because the virtual machine relies on subtype information to determine which exception handler it needs to invoke, flattening an application's exception classes may cause the virtual machine to invoke the wrong exception handler, or even no exception handler at all. Because we cannot modify the virtual machine from our application, we modify the application to take over part of the work of the virtual machine. That is, we rewrite the exception handlers of each of the application's methods to handle exceptions of all possible types. Then, using a series of gotos, and instanceof checks on the thrown exception, we redirect the flow of control to code that can handle the exception.

Before we go into more detail on how our algorithm works, it is important to first explain how exception handling is implemented in bytecode. In bytecode there are no instructions that correspond to the try and catch primitives in source code. Instead, code is laid out as if there were no try blocks, and gotos are inserted to redirect control around the code in the catch blocks, as shown in Figure 2.6(b). The actual exception handling information is stored in an exception table. Each row in this table corresponds to a single handler and contains the range of instructions it covers, the offset of its first instruction, and the type of exceptions it handles. Handling an exception using the information in this table is straightforward. Whenever an exception occurs, the virtual machine looks for the first entry in the table for which the offset of the instruction that caused the exception is in the range $[from, to[$, and for which the exception is an instance of the handler's type. If such an entry is found, control is transferred to the instruction at the target offset.

Given the bytecode representation (or a similar intermediate rep-

from	to	target	type
4	5	6	ExceptionA
4	5	8	ExceptionB
3	4	11	ExceptionC
4	5	11	ExceptionC
5	10	11	ExceptionC

Table 2.3: Exception table with split entries.

resentation) of a method, our algorithm first splits the entries in the method's exception table into multiple entries in such a way that the ranges of any two entries either do not overlap, or fully overlap. The result for the exception table in Figure 2.6(b) is shown in Table 2.3.

Then, to create the method's new exception table E, our algorithm traverses the split exception table S obtained in the previous step from top to bottom. For each entry $(from, to, target, type) \in S$ it performs the following actions.

1. Let e be the handler in E that handles exceptions in the range $[from, to[$. If such a handler does not exist, create it, and register it as a handler for exceptions of type java.lang.Throwable.
2. Add code to e that checks whether the caught exception is an instance of $type$, and if it is, redirects the control flow to the instruction at offset $target$.

Finally, our algorithm appends to the code of each handler in E code that rethrows the caught exception, such that it can be handled by the calling method when the current method does not handle it. Note that rethrowing the exceptions does not cause problems, as their stack traces are not modified; the stack traces of exception objects are filled in when they are created, not when they are thrown.

The result for the method in Figure 2.6(b) is shown in Figure 2.7. In this case the algorithm has created three new exception handlers, starting at lines 14, 21, and 24, respectively. The first one handles exceptions of type ExceptionA, ExceptionB, and ExceptionC (in that specific order). The other two only handle exceptions of type ExceptionC. Note that for this simple example it is not strictly necessary to generate separate code for the last two handlers. Our algorithm could have simply set the target of these handlers to the instruction on line 18. However, in practice

```
 1 public void m(){
 2   stmt0
 3   stmt1
 4   stmt2
 5   goto 9
 6   handler0
 7   goto 9
 8   handler1
 9   stmt3
10   goto 12
11   handler2
12   stmt4
13   return
14   if ex instanceof ExceptionA
15     goto 6
16   if ex instanceof ExceptionB
17     goto 8
18   if ex instanceof ExceptionC
19     goto 11
20   throw ex
21   if ex instanceof ExceptionC
22     goto 11
23   throw ex
24   if ex instanceof ExceptionC
25     goto 11
26   throw ex
27 }
```

Exception table:

from	to	target	type
4	5	14	Class java/lang/Throwable
3	4	21	Class java/lang/Throwable
5	10	24	Class java/lang/Throwable

Figure 2.7: Method in Figure 2.6(b) after rewriting its exception handlers.

such optimizations may be difficult, especially when try blocks have multiple associated catch blocks for different types of exceptions, or when a method's exception handlers have been transformed to obfuscate control flow or to thwart decompilers [12].

Chapter 3

Interface Merging

In the code in Figure 2.4 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 Figure 3.1 and Figure 3.2. 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.

3.1 Interface Partitioning

Other than to avoid excessive code size overhead, there is another reason for which we cannot merge all interfaces into a single super interface.

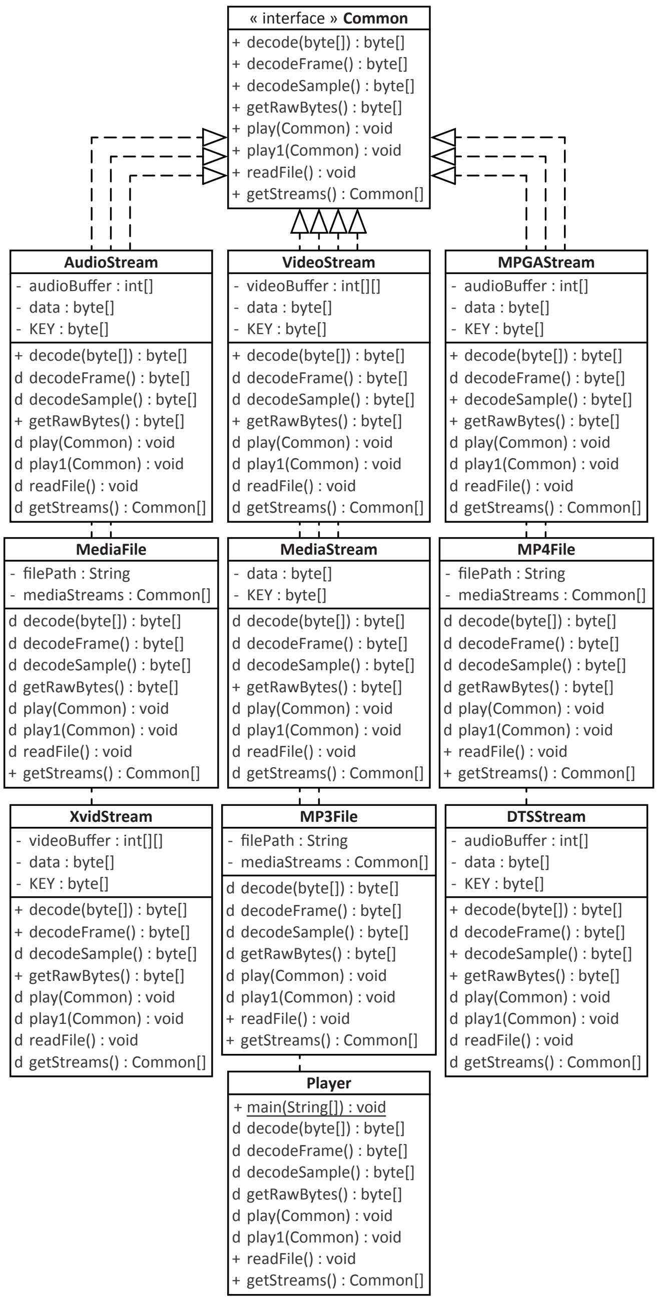


Figure 3.1: Class hierarchy of the flattened media player after interface merging.

```
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 (myCheck.isInst(0, 1, ms.getClass()))
          player.play(ms);
        else if (myCheck.isInst(1, 1, ms.getClass()))
          player.play1(ms);
  }
}
public class MP3File implements Common {
  public void readFile() {
    InputStream inputStream = ...;
    byte[] data = new byte[...];
    inputStream.read(data);
    Common as = new MPGAStream(data);
    mediaStreams = new Common[]{as};
    return;
  }
}
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 byte[] decode(byte[] data){ ... }
}
```

Figure 3.2: Partial implementations of the Player, MediaStream and MP3File classes after class hierarchy flattening and interface merging. Changes compared to the code after flattening are shown in red.

Interface merging (IM), like class hierarchy flattening, 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 the merging to sets of interfaces of which the subtrees originate from within the same directories or archives. The merged interface can then be packaged in those same directories and archives, 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 [44] 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.

3.2 Interface Merging

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

1. Create a new interface k that declares all methods declared in all interfaces in I_j.
2. Add k to the directory or archive that contains the interfaces in I_j.
3. For each class c in the application, replace all references to interfaces in I_j by references to k. If c implements one of the interfaces 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.

Figure 3.1 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 dissertation, 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 Figure 3.1, we merge play, decodeSample, decodeFrame, and readFile into merged1, and getStreams and play1 into merged2,

```
public class Player implements Common {
  public byte[] merged1(Common as) {
    /* send as.getRawBytes() to audio device */
  }
  public Common[] merged2(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 (myCheck.isInst(0, 1, ms.getClass()))
          player.merged1(ms);
        else if (myCheck.isInst(1, 1, ms.getClass()))
          player.merged2(ms);
  }
}
public class MP3File implements Common {
  public byte[] merged1() {
    InputStream inputStream = ...;
    byte[] data = new byte[...];
    inputStream.read(data);
    Common as = new MPGAStream(data);
    mediaStreams = new Common[]{as};
    return data;
  }
}
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 byte[] decode(byte[] data){ ... }
}
```

Figure 3.3: Partial implementations of the Player, MediaStream and MP3File classes after class hierarchy flattening, interface merging, and method merging. Changes compared to Figure 3.2 are shown in red.

as shown in Figure 1.3 and in Figure 3.3. As a result 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, it selects sets of related methods that can be merged. Next, it selects increasingly more expensive combinations of these sets and merges them greedily.

3.3 Method Set Selection

Changing the signature of a method requires making similar changes to the signature of overridden and overriding methods [73]. 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 $\mathbb{T} = \mathbb{A} \cup \mathbb{L}$ the set of all classes and interfaces.
- Let $M : \mathbb{T} \mapsto \mathbb{M}$, with $M(t)$ all methods declared in t.
- Let $S : \mathbb{M} \mapsto \mathcal{P}(\mathbb{M})$ the function that returns the set of methods $S(m)$ that should have the same signature as m.
- Let $f : \mathbb{M} \mapsto \{false, true\}$, 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 whose signature can be changed.

  do we do this?

The set of method sets that can be merged can now be computed as

$$\mathbb{S} = \{S(m) \mid \exists\, i \in \mathbb{I} \,.\, m \in M(i) \wedge \forall\, n \in S(m) \,.\, f(n)\}.$$

In what follows we extend the notions of signature, return type, name, and parameter type list from single methods to sets of methods in $\mathbb{S}$. This makes sense, because each $S \in \mathbb{S}$ is a set of same-signature methods.

3.4 Method Set Merging

Several constraints limit method merging. For example, since a method can only declare one return type, two methods with different non-void return types cannot be merged. Furthermore, we must avoid merging

Algorithm 3.1: Greedy method merging.

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

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, it is possible. Consider for instance the flattened class hierarchy in Figure 2.2, in which the transformable class B does not implement one of the subtree interfaces. Merging I_2 and I_3 into I results in the hierarchy of Figure 3.4. In this hierarchy, 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 for a method m if c or one of its superclasses has an implementation for m. In the class hierarchy of Figure 3.4, 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 greedy method merging algorithm shown in Algorithm 3.1. In this algorithm, $C : \mathcal{P}(\mathbb{S}) \mapsto \mathbb{R}^+$ 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. Given a set $\mathfrak{s} = \{S_i, S_j\} \subset \mathbb{S}$ that adheres to the merge condition, it performs the following steps.

1. Compute the signature $s = {<}r, n, p{>}$ for the merged methods.
2. Create an empty set $\hat{\mathfrak{s}}$ that will hold the merged methods.
3. Compute for each class c the set $N(c)$ of methods to merge as

$$N(c) = \{m \in M(c) \mid \exists\, S_k \in \mathfrak{s} \,.\, m \in S_k\}.$$

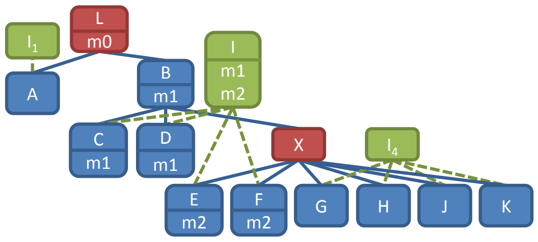


Figure 3.4: Class hierarchy after merging I_2 and I_3. Dummy methods introduced by merging have been omitted for clarity.

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

In step 1 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.

Chapter 4

Object Factory Insertion

Even after interface merging some statements still expose type information. For example, after the allocation on line 9 of Figure 3.2 the player variable points to an object of type Player. From this information, points-to analyses can deduce points-to sets of many local variables. In turn, other analyses that construct call graphs [39, 72] or compute program slices [70, 79] will also regain some precision to the advantage of attackers. For instance, knowing the exact type of the object bound to the player variable allows an attacker to resolve the calls on lines 14 and 16 of Figure 3.2 to methods defined in class Player.

To prevent the propagation of precise type information from allocation sites, we replace allocations by calls to object factories [36] that can 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 (CHA) [29], which as a result of CHF and IM will not be very meaningful. To achieve this effect the transformation proceeds in two steps. First, it preprocesses the program to maximize the effectiveness of the next step. Then, it creates the object factories and replaces object creation expressions by calls to those factories.

4.1 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, the return type of a factory must be assignment-compatible with the variable or field to which the returned object is assigned. In our example MyFactory.create() therefore has to return an object whose type is equal to or a subtype of Common. To enable the 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 [35], because more concrete types can aid analyses such as CHA [29] 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 Figure 3.4 is implemented as in Figure 4.1(a). The types of x1 and x2 were not changed by CHF or IM because these variables are of the non-transformable 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 does not change, as m2 is defined in I. Figure 4.1(c) shows the resulting code.

In the next section we give an overview of Gagnon et al.'s algorithm, and we discuss in more detail how we modified it to fit our needs.

4.1.1 Inferring Abstract Types

The type inference algorithm by Gagnon et al. [35] was designed to operate on a 3-address intermediate representation of a program's bytecode, called Jimple [74]. Overall, Jimple code is closely related to Java source code. One important concept is expressed differently in both representations, however. In Jimple, object creations are implemented using two different instructions, like in bytecode, whereas in Java source code an object is created with a single statement. Figures 4.1(a) and 4.1(b) illustrate this difference. Note that in Jimple each allocation of the form

```
public void m1() {
  X x1 = new X();

  x1.m0();
  X x2 = new X();

  x2.m1();
  I c = new C();


  c.m2();
}
```

(a)

```
public void m1() {
  X x1 = new X
  x1.<X: void <init>>()
  x1.<L: void m0>()
  X x2 = new X
  x2.<X: void <init>>()
  x2.<B: void m1>()
  C c_ = new C
  c_.<C: void <init>>()
  I c = c_
  c.<I: void m2>()
}
```

(b)

```
public void m1() {
  L x1 = LFactory.create(key1);

  x1.m0();
  B x2 = BFactory.create(key2);

  x2.m1();
  I c = IFactory.create(key3);


  c.m2();
}
```

(c)

Figure 4.1: Example of object factory insertion. (a) Original code. (b) Simplified typed Jimple code. (c) Code with factories.

new Y must be followed by a call to one of Y's constructors, as indicated by the calls to Y: <init>. In Java source code these two statements are combined into a single constructor call.

Note that the code shown in Figure 4.1(b) is actually typed Jimple code, as each local variable is already assigned a type. Typed Jimple code is what almost all Jimple transformations, including our obfuscations, operate on[1]. However, whether or not Jimple code is actually typed does not matter for type inference algorithms, because they were designed to work on untyped Jimple code anyway. Since they cannot rely on type information from local variables in untyped Jimple code, they use different information to infer a type for each variable. Our obfuscations can hence simply invoke type inference algorithms on the code they are operating on, without first converting that code to untyped Jimple.

To compute the types of local variables of (un)typed Jimple code, the type inference algorithm by Gagnon et al. builds a system of type constraints, which it expresses as a graph. Each graph has three different types of components: *hard nodes*, *soft nodes*, and directed edges connecting those nodes. Hard nodes represent explicit types, while soft nodes represent variables. Each edge $a \leftarrow b$ in the graph represents a relation that indicates that b should be assignable to a. The graph itself is created by iterating over the Jimple instructions and adding edges between hard and soft nodes as implied by the instructions. Once the graph is constructed, the type inference problem is solved by merging hard and

[1]The only exceptions are type inference algorithms and a handful of optimizations.

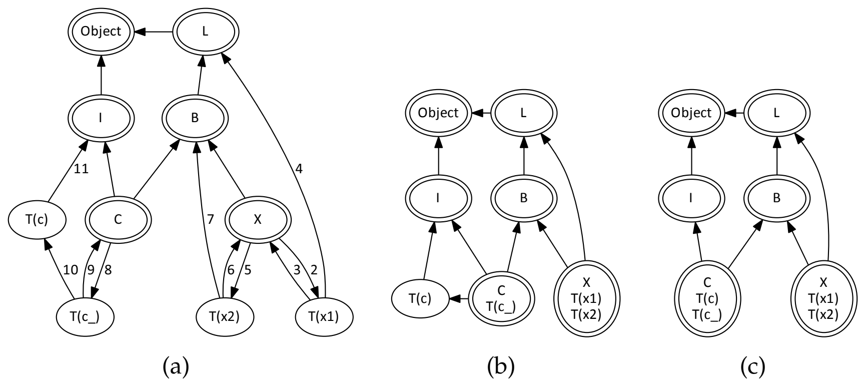


Figure 4.2: Type inference graphs constructed using the algorithm by Gagnon et al. [35]. Soft nodes are circled once, hard nodes twice. (a) Initial graph. (b) Graph after collapsing connected components. (c) Final graph.

soft nodes. A solution to the type constraint problem has been found when each soft node is merged with a hard node, and hence when each local variable is assigned a type[2].

Hard and soft nodes are merged in three steps. First, the algorithm computes the set of connected components (i.e. cycles) in the graph. Each component is then collapsed into a single node to turn the graph into a DAG. Next, all transitive constraints, with the exception of hard node to hard node constraints (which represent the type hierarchy), are removed from the DAG to simplify it. Finally, nodes that have only one parent or child constraint are simplified. Figure 4.2 illustrates the process.

Figure 4.2(a) shows the initial type inference graph for the code fragment in Figure 4.1(b). By convention each soft node representing a variable a is labeled T(a), which corresponds to the currently unknown type of the variable. Furthermore, for convenience we have labeled each edge with the line number of the Jimple instruction that implies the corresponding constraint. The graph contains two connected components consisting of $\{\mathsf{T(c\_)}, \mathsf{C}\}$ and $\{\mathsf{T(x1)}, \mathsf{T(x2)}, \mathsf{X}\}$, respectively. Both components contain a single hard node, which means they can each be collapsed into a single hard node as shown in Figure 4.2(b). Since the resulting graph does not contain any transitive constraints that involve

[2]In some cases the algorithm may not be able to merge certain hard and soft nodes. It then fails to determine a type for some variables. The authors provide details on how to deal with such cases. However, since they only account for 0.2% of all cases in practice, we do not discuss them here.

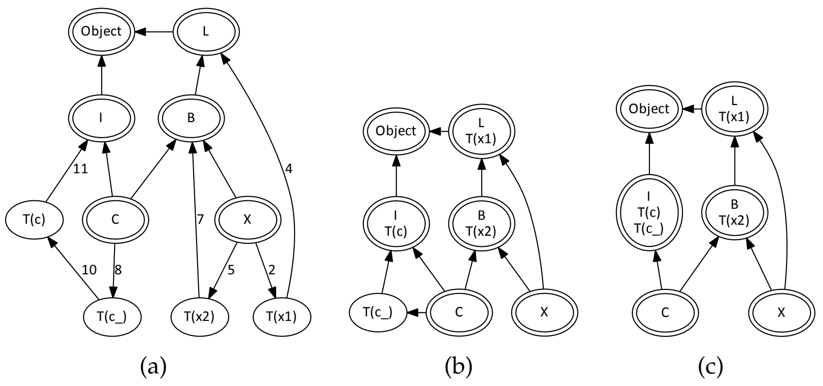


Figure 4.3: Type inference graphs constructed using our modified algorithm. (a) Initial graph. (b) Graph after first round of merging. (c) Final graph after second round of merging.

soft nodes, we can skip the second step of the algorithm. In the final step, the last remaining soft node T(c) is merged with its single child constraint node "C T(c_)". As stated by the authors, merging with child nodes results in more precise types for the variables.

However, this is not what we want. We want variables to have more abstract types. To achieve this, we modify the algorithm in two ways. First, we ensure that soft nodes are merged with their parents, and not their children, whenever possible. Second, we also modify how the type inference graph is constructed to reduce the number of connected components that involve variables used in object creations. This is desirable because the algorithm generally assigns very precise types to those variables. If, by contrast, we ensure that there is no connected component that contains those variables, we can assign them a more abstract type.

Because of the way object creations are expressed in Jimple, all variables that hold objects from the point they are allocated (with a new expression) until they are initialized (with an <init> call) will always be part of a connected component. Furthermore, that component will also always include a hard node representing the type of object that is created: Each definition x = new X adds a constraint of the form $\mathsf{T(x)} \leftarrow \mathsf{X}$, while each call a.<X:<init>>(...) uses x as a variable of type X, and hence adds the constraint $\mathsf{X} \leftarrow \mathsf{T(x)}$ to the type inference graph. As a result, both the lower and the upper bound for T(x) are equal to X, which means X is the only possible type that can be assigned to x.

To have more freedom in selecting a new type for each variable, we construct the type inference graphs without taking into account the <init> calls. This is not a problem since these calls will be replaced by factory calls anyway. Figure 4.3(a) shows the resulting graph for the code in Figure 4.1(b). In this graph there are no connected components, so the nodes for T(c_), and T(x1) and T(x2) are not merged with those for C and X, respectively. There are also no transitive constraints that can be simplified, only single constraints.

Our modified algorithm simplifies single constraints with the following prioritized set of rules:

1. Merge y with x if y is a soft node, x is a hard node, and $x \leftarrow y$ is the only parent constraint for y.

2. Merge y with $hcd(y)$ if y a soft node of which all parents are hard nodes, and $hcd(y)$ is unique.

3. Merge y with x if both x and y are soft nodes, and $x \leftarrow y$ is the only parent constraint for y.

In rule 2 we compute $hcd(y)$ as follows. Let H be the set of all highest common descendants of y's parents, and let C be the set of all hard nodes from which y can be reached. $hcd(y)$ is unique and equal to h if h is the only type in H for which it holds that $\forall\, c \in C \,.\, h \in t^s(c)$.

Rule 3 is the same as in the original algorithm, whereas rules 1 and 2 can be seen as the complement of their corresponding rules in the original algorithm. However, technically our version of rule 2 is more generic than the original, which only applies in case all hard nodes represent class types. Our version also works when the hard nodes represent interface types. In that case there is a likelihood that the highest common descendant is not unique. However, by also taking into account the hard nodes from which a node can be reached (which correspond to definitions in Jimple code), our algorithm often finds a unique solution.

Given the graph in Figure 4.3(a), rule 1 can be applied to T(x1), T(x2), and T(c) to yield the graph in Figure 4.3(b). In this graph T(c_) now also has a single hard parent, which means rule 1 can be applied once more. The resulting graph is shown in Figure 4.3(c), which represents the same type assignment as the one used in Figure 4.1(c).

It is important to note that our algorithm will not be able to fully simplify all type inference graphs. When tested on the applications from the DaCapo benchmark suite [14], our algorithm fails for around 0.5% of all methods. The reason is that many of the DaCapo benchmarks contain references to types for which neither they, nor their libraries (including the Java standard library) contain a definition. Since little is known about these types and their position in the type hierarchy, our algorithm cannot create hard nodes for them, and it fails. Note that the fact that the algorithm fails is not a result of the changes we made to it; the original algorithm by Gagnon et al. faces the same problem. However, in practice it is not really an issue when our algorithm fails. In those cases we can simply use the original types of the variables.

4.2 Object Factory Creation

In this step OFI replaces object creation expressions by calls to object factory methods. For each object creation o : $\mathsf{x} = \texttt{new}\ \mathsf{C}(\ldots)$, where $\mathsf{C} \in \mathbb{A}$, and the declared type of x is X, it performs the following steps.

First, it determines the properties of the factory method. In doing so it constructs $U = \{u \in t_s(\mathsf{X}) \cap t^s(\mathsf{C}) \mid u \in J_\mathsf{C}\}$, where J_C is the jar or archive that contains class C. Given U, it computes the return type of the factory method as

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

with $q : \mathbb{A} \mapsto \mathcal{P}(\mathbb{A})$, and $q(u) = \{d \in \mathbb{A} \cap t_s(u) \mid d \in J_\mathsf{C}\}$.

It may seem strange to compute R in this manner, or even that R has to be computed at all, as X clearly is the best choice for the return type. After all, this type is the most abstract type that could be assigned to x. As a result, it has more subtypes that any other type in $t_s(\mathsf{X})$, and is hence the best choice when creating factory methods that must be able to return many different types of objects.

However, it is not always possible to choose X as the return type, because doing so may lead to class loading errors. By default, we can only assume that when the object creation o is executed, class C has been loaded. Or, more importantly, this means that there is a class loader that, before executing o, can load class C, and other classes located in the same directory or archive as C[3]. Hence, when o is replaced by a call to a factory method, we must ensure that the return type of the factory

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

```
public class Player implements Common {
  public byte[] merged1(Common as) {
    /* send as.getRawBytes() to audio device */
  }
  public Common[] merged2(Common vs) {
    /* send vs.getRawBytes() to video device */
  }
  public static void main(String[] args) {
    Common player = CommonFactory.create(...);
    Common[] mediaFiles = ...;
    for (Common mf : mediaFiles)
      for (Common ms : mf.getStreams())
        if (myCheck.isInst(0, 1, ms.getClass()))
          player.merged1(ms);
        else if (myCheck.isInst(1, 1, ms.getClass()))
          player.merged2(ms);
  }
}
public class MP3File implements Common {
  public byte[] merged1() {
    InputStream inputStream = ...;
    byte[] data = new byte[...];
    inputStream.read(data);
    Common as = CommonFactory.create(...);
    mediaStreams = new Common[]{as};
    return data;
  }
}
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 byte[] decode(byte[] data){ ... }
}
```

Figure 4.4: Partial implementations of the fully type-obfuscated versions of the Player, MediaStream and MP3File classes. Changes compared to Figure 3.3 are shown in red.

method, the class declaring the factory method, and the declaring classes of constructors called by the factory method are located in the same directory or archive as C, such that they can be loaded by the same class loader that loads C. The return type is therefore chosen as the type that is assignment-compatible with X, that is located in the same directory or archive as C, and that has the most subtypes in that directory or archive. The latter condition ensures that the set of classes whose constructors the factory should be able to call is as large as possible.

From R, the OFI algorithm computes the set of constructors that will be called by the factory method as

$$K = \{m \in M(d) \mid d \in q(R) \wedge m \text{ is a constructor}\}.$$

Second, given R and K, let RKFactory be the factory class with a method create with return type R that creates objects using the constructors in K. If this class does not exist, OFI creates one in J_{C}. The parameter list of create consists of the combined parameter 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 Figure 4.1(a), our algorithm creates LFactory, BFactory, and IFactory and rewrites the allocations as in Figure 4.1(a). Without preprocessing, it 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 twelve and ten different types, respectively.

Technically, LFactory requires U to also include library types, which it does not.

For the media player, Figure 4.4 shows the resulting code.

Chapter 5

Evaluation

In this chapter we give an overview of the experimental results obtained using our type obfuscations. We also discuss some of the limitations and restrictions of our obfuscations, address their correctness, and go into the effects they have on reliability and maintainability of the transformed programs. Finally, we compare our obfuscations to related work.

As part of the evaluation we implemented class hierarchy flattening, interface merging, and object factory insertion in Soot 2.5.0 [46, 74], 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 several 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 offers a series of refactoring transformations, including *encapsulate field*, *rename field-/method*, and variations of *push down field/method* and *extract interface* that are required to implement CHF, IM and OFI [34]. In our proof-of-concept tool, the dummy method bodies are empty.

5.1 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 [16]. 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 two restrictions on applications eligible for our type obfuscations. First, each class loader only loads classes and interfaces of which the definition is known at obfuscation time. Second, each class loader (i) only loads classes from a fixed set of directories or archives, and (ii) is able to load new classes and interfaces from those directories and archives. With these restrictions we can determine exactly in which directory or archive to insert new classes and interfaces such that they are loaded by the correct class loader.

Furthermore, it is important to note that 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.

5.2 Benchmarks

We used the DaCapo 9.12 benchmark suite [14] 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 5.1. 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 are 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

Benchmark	# application classes	# application interfaces	# transformable classes (CHF)	# jar files	code size (MB) pre IO	code size (MB) post IO
avrora	1836	83	1657 (90%)	2	4.1	2.9
batik	3787	856	3383 (89%)	7	12.5	9.3
eclipse	5213	1261	3886 (75%)	49	25.7	17.2
fop	4033	446	3105 (77%)	8	11.0	8.8
h2	1843	78	1454 (79%)	5	9.3	7.0
jython	3702	166	941 (25%)	8	11.8	10.6
luindex	605	28	510 (84%)	4	1.9	1.2
lusearch	608	28	510 (84%)	4	1.9	1.2
pmd	1999	451	1508 (75%)	7	5.6	4.4
sunflow	679	59	557 (82%)	3	2.0	1.6
tomcat	2173	268	1538 (71%)	27	10.1	7.1
xalan	2460	426	2111 (86%)	6	9.6	7.6

Table 5.1: Overview of DaCapo 9.12-bach benchmarks before and after Identifier Obfuscation (IO).

code generation, we cannot transform the static jython classes referenced by the dynamically generated classes either.

The two benchmarks not shown in Table 5.1 are tradebeans and tradesoap, which are based on the Apache DayTrader J2EE workload[1]. We are uncertain whether these applications meet the requirements for transformation, because we did not invest time in analyzing their class loaders. The reason for this is twofold. Not only is the class loader behavior of these benchmarks more complex than for the others, they also contain race conditions that may or may not get triggered depending on the code layout. Hence, even if these benchmarks would meet the requirements, we would still not be able to transform them reliably.

As a baseline for comparison, we use the original DaCapo bytecode, but with identifier names obfuscated [21]. 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

[1]http://geronimo.apache.org/GMOxDOC22/daytrader-a-more-complex-application.html

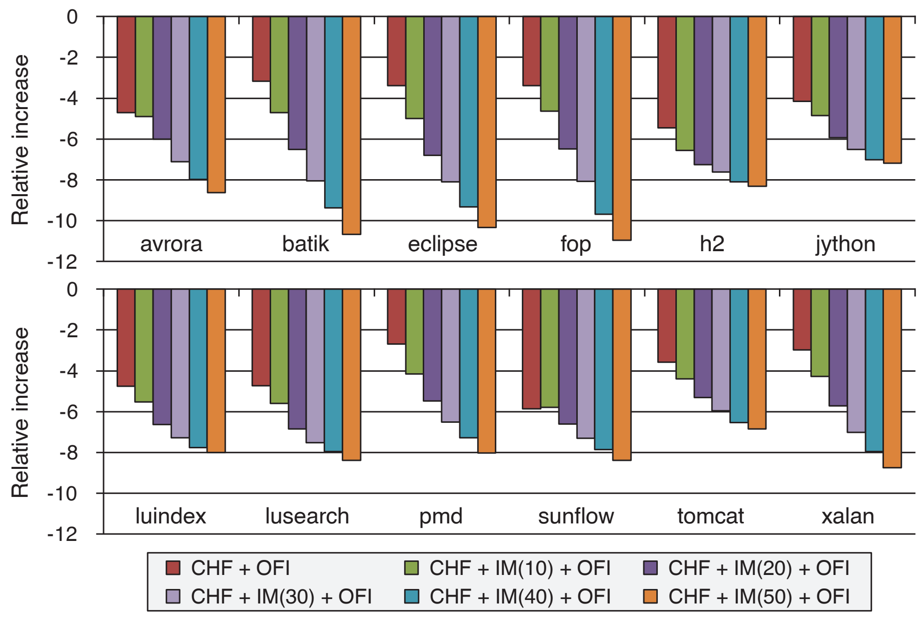


Figure 5.1: QMOOD understandability

and MM. We report the average results for those ten versions.

5.3 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 by Bansiya and Davis [10]. 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 program's design, including abstraction, encapsulation, coupling, cohesion, polymorphism, complexity, and design size [10]. 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 [10], less understandable versions will have lower scores. Figure 5.1

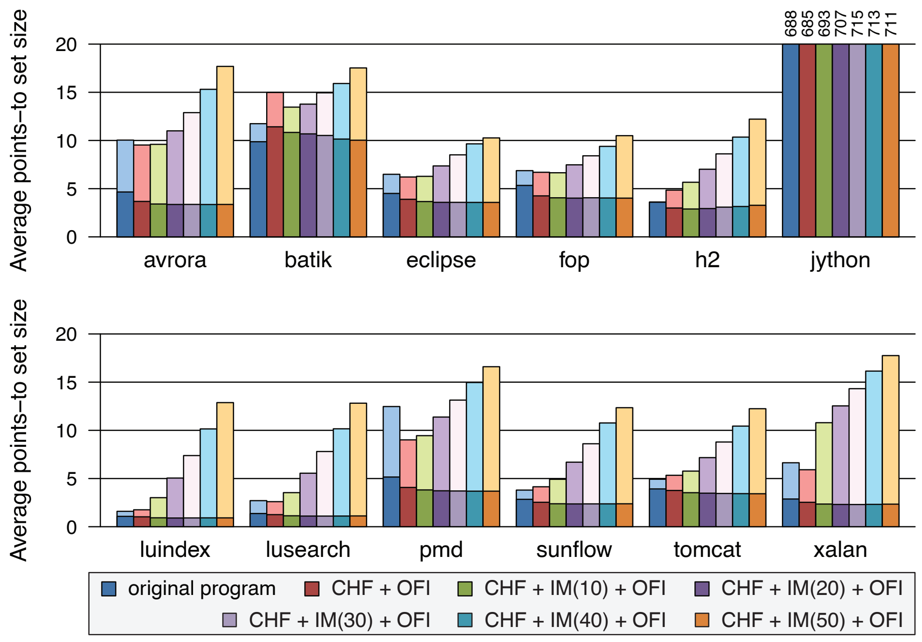


Figure 5.2: Average points-to set sizes. Dark bars denote sizes obtained without OFI, light bars denote sizes with OFI.

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 5.1. 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, Figure 5.3 shows the relative contribution of the different QMOOD metrics to lusearch's understandability, for the same program versions as in Figure 5.1. 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 Figure 5.1, this chart shows that most of the understandability reduction results from increases in

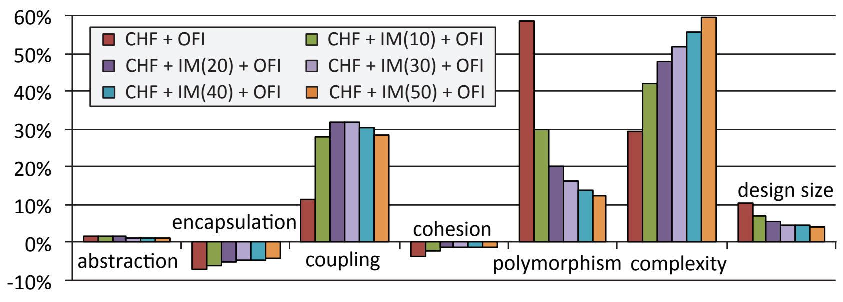


Figure 5.3: Contribution of design metrics to lusearch's understandability score.

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 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 analyses because the constructed call graphs become bigger. Hence, reducing the precision of points-to analyses by causing them 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 robust and configurable T.J. Watson Libraries for Analysis (WALA, http://wala.sf.net) to compute points-to sets using simple class hierarchy analysis [29], context-insensitive 0-CFA [40], and the partially context-sensitive, 0-1-CFA and 0-1-container-CFA of WALA. Here 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 [16]. Figure 5.2 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 the 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.

One way to increase the points-to set sizes of a method's added parameters is to have the method's callers pass objects bound to their local variables, instead of just null dummy arguments. However, the calling methods may not always have references to objects of the correct type, and even if they do, these references may not always be available at the point that the method that requires them is invoked. Alternatively, new dummy objects to be passed as arguments can also be created directly, or by means of object factories. However, in those cases the run-time overhead of method invocations may increase significantly.

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 (excluding jython) with a factor 4.67 on average, ranging from a factor 1.78 for batik to a factor 11.98 for luindex.

5.4 Overhead

The dark bars in Figure 5.4(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 Figure 5.4(b,c) include all 10 runs of the benchmarks in their harness. This includes the warm-up runs during which the JIT compiler is very active. Figure 5.4(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 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 [55]. For some bench-

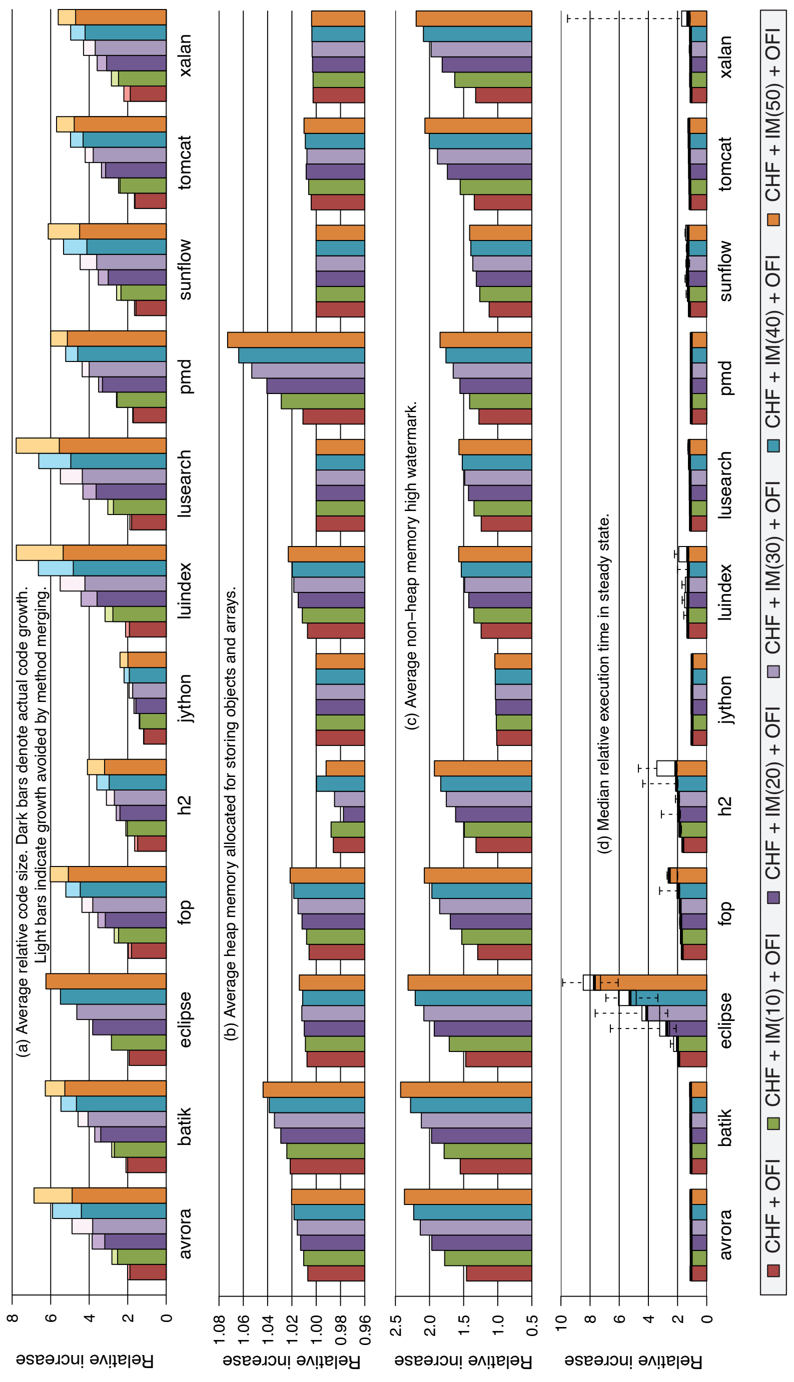


Figure 5.4: Overhead of the obfuscations. All metrics are relative to the original benchmarks.

marks, like h2, the escape analysis accidentally performs better on some more heavily obfuscated versions.

Figure 5.4(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 overhead of our obfuscations is hence limited.

In contrast to 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 Figure 5.4(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 much potential for reducing the overhead of our obfuscations by making the currently partially randomized and unbounded application of merging more tuned, controlled, and limited. In Chapter 6 we go into more detail on how this can be achieved.

5.5 Correctness, Reliability and Maintainability

A formal validation of CHF, IM, and OFI is out of the scope of this dissertation. 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 conference paper [32], as well as intermediate program versions generated by the individual transformation steps discussed in chapters 2, 3, and 4. All of these experiments were stressing Soot, WALA, and TamiFlex beyond their preexisting 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 obfuscation. As for the TamiFlex Play-out Agent and Booster, we point to the literature for a discussion of their validity [16].

As mentioned earlier, our obfuscations are quite similar to existing refactorings [34, 68, 73]. Tip et al. express the valid refactoring space by means of type constraints [73]. 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 declarations. These constraints determine the freedom to alter declarations in the program without affecting type correctness and without changing the program's functionality. This is done 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, including the rewritten cast operations and merged methods, meets all requirements for maintaining program behavior, i.e., that at all places and at all times, the same exceptions will be thrown as in the original program, and that the same or equivalent (e.g., merged) methods are invoked.

Finally, the class loader restrictions imposed in Section 5.1 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 do not 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 [38] remain intact.

Besides providing obfuscation and introducing overhead, our transformations come with some important side effects. Firstly, 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 traces back from the obfuscated to the original code.

Secondly, as our transformations alter the execution speed of different code fragments differently, they may expose or hide race conditions in multi-threaded code. In this regard, our transformations do not differ from other static or JIT code optimization, or 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 may require more bandwidth.

5.6 Comparison to Related Work

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 [66]. 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. Furthermore, their evaluation does not contain any criteria related to software protection, software understandability, or software complexity. Additionally, they mostly attribute limitations to the applicability of their transformations to immaturity of their tool, instead of discussing more fundamental issues. By contrast, we proposed transformations that from the very start maximally remove the class hierarchy, and of which their overhead in terms of code size, memory footprint, and performance, as well as their impact on program understandability are evaluated for a set of large real-life programs. Furthermore, rather than being immature, our prototype tool pushes the application of our obfuscations to the fundamental limits relating to external libraries, dynamic class loading and reflection.

CHF can be combined with class coalescing. In particular, CHF enables more efficient coalescing. Coalescing MP3File and VideoStream in Figure 1.1 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. [22] 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. CHF can prepare a program for false factoring [22]. In Figure 1.3 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. [62] 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 object factory insertion transformation 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.

The Java Binary Enhancement Tool (JBET) developed as part of the Self-Protecting Mobile Agents (SPMA) project [9, 28] implements a series of techniques that break down semantically rich Java structures such as classes, method invocation, virtual method dispatch, exception handling, data representations, and garbage collection. In the most extreme case, JBET is able to collapse an entire program into a single method that contains all the code in the application, that only constructs objects of a single Memory class representing a generic data structure, and that implements its own mechanisms for dynamic method dispatch, exception handling, and garbage collection, all while staying within the Java bytecode domain.

As part of its obfuscation routine JBET uses function flattening [76–78] to flatten all the methods in the program, after which it merges them into a single flattened method to hide the boundaries between the original methods. Additionally, JBET expresses instances of the original program's classes using instances of class Memory, such that the original classes can be removed. Instances of the Memory class are also used to implement virtual method tables for dynamic method dispatch, and to implement activation records to store local variables, and to pass data between methods after these have been flattened and merged.

In general, JBET is able to obfuscate more type information than the transformations presented in this thesis, but at significant overheads. Programs transformed using JBET can become up to 10 times larger, and between 4 and 20 times slower. The main difference between our techniques and the ones implemented in JBET is that our techniques try to get rid of as much type information as possible while still relying on fundamental concepts such as method and classes, and on the mechanisms for dynamic method dispatch, exception handling, and garbage collection provided by the run-time system. Programs transformed using JBET implement many of these concepts and mechanisms themselves, which may explain the rather large overheads[2].

Even though JBET's transformations differ significantly from CHF, they are also able to remove a program's class hierarchy, albeit by getting rid of the program's classes and by representing their instances as Memory objects. However, an important drawback of JBET is that transformed programs implicitly encode information about the original type hierarchy in the tables that are used to implement virtual dispatch. Whenever a class B in the original program inherits a method from its superclass A, the virtual tables associated with the Memory instances representing the instances of classes A and B will contain the same entry for that method. The authors of JBET acknowledge this fact and even state that their automatic deobfuscator is able to exploit it to recover information about a program's original class hierarchy. In general, we expect that setting up a similar attack against CHF will be more difficult because CHF copies methods from superclasses to their subclasses, after which the copies are treated as separate methods during obfuscation. This means that finding related classes will not be as simple as finding virtual call tables that contain the same entries.

Snelting and Tip [64, 65] 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

[2]JBET is not available for experimentation, so it is difficult to ascertain what the exact causes of overhead are.

to split classes. This analysis has been extended and implemented in the refactoring tool KABA [68]. 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 SandMark[3] and Stigmata[4]. 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.

Malicious users may also resort to dynamic techniques to attack programs obfuscated using our techniques. To defeat object factory insertion an attacker could instrument the application under attack to log which constructor is invoked at each factory call site. Given enough coverage he or she may be able to replace all calls to object factories by calls to the original constructors, and completely remove the factory classes from an obfuscated program. This is possible because at each object creation site in the original program only one constructor is invoked, and because OFI preserves this property by replacing each constructor call by a call to a factory method that invokes that constructor.

After successfully defeating OFI, an attacker can try to split the large interfaces created during IM into smaller ones that are implemented by fewer classes. To this end he or she may take advantage of the fact that even though many variables, parameters, and fields are of generic interface types and can theoretically store objects of many different types, in practice many of them will only store objects of the same (small) set of types as they did in the untransformed application[5]. As a result, information on which types of objects are stored in the same variables, parameters, and fields, together with information on which classes implement the same set of interfaces and inherit from the same (library) superclasses may help partition classes into related sets and give insights into how to best split the large interfaces created during IM.

[3]http://sandmark.cs.arizona.edu

[4]http://stigmata.sourceforge.jp

[5]Figure 5.2 shows a large difference in points-to sets sizes between program versions depending on whether or not OFI has been applied.

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 Chapter 1. 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 [21–23, 51].

Chapter 6

Reducing Obfuscation Cost

Like many other obfuscating transformations, class hierarchy flattening, interface merging, method merging, and object factory insertion come at a certain cost in terms of code size and execution time of the transformed programs. Even though these obfuscations are effective in terms of protection, they use rather naive cost functions that poorly reflect the actual changes in code size and execution time. As a result, some transformed applications became up to six times larger and up to eight times slower.

For local transformations that, e.g., insert opaque predicates [23], this cost can be managed easily. The overhead in terms of code size can simply be computed from the size of the predicate and the number of times it was inserted. Additionally, profile information can be used to avoid inserting the opaque predicates in frequently executed code sections. However, for global transformations such as ours, it is much more difficult to keep track of potential code size and execution time overhead during transformation, as many classes are involved in each of the different transformation steps. Yet, failing to keep track of this information accurately can result in programs that are unacceptably large or slow, which may limit the practical usefulness of the transformations. To avoid this, we present improved versions our transformations that can accurately estimate the impact of different obfuscation steps on the size and execution time of programs, and act accordingly. For these improved transformations, our results not only show significant reductions in overhead, but also that these reductions can be obtained with limited impact on the level of protection offered by the transformations.

The remainder of this chapter is organized as follows. First, Section 6.1 addresses several issues with our transformations as presented in previous chapters, and motivates the need for improvements. Next, Sections 6.2 and 6.3, and 6.4 present improved versions of method merging and object factory insertion, respectively. Finally, we evaluate the proposed optimizations in Section 6.5.

6.1 Motivation

We illustrate the problems with class hierarchy flattening, interface merging, method merging, and object factory insertion using an example program of which part of the code is shown in Figure 6.1(b). The class hierarchy of the application is shown in Figure 6.1(a). It consists of two subtrees of java.lang.Object, rooted at X1 and Y1, respectively.

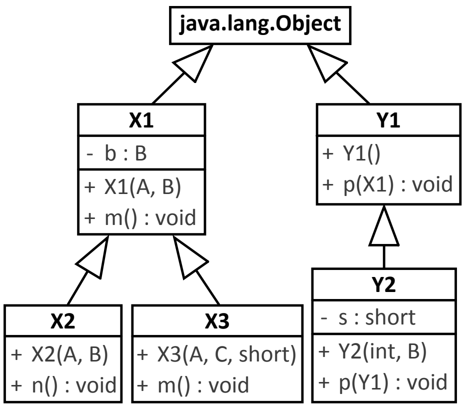
6.1.1 Class Hierarchy Flattening

The class hierarchy flattening transformation described in Chapter 2 removes subtype relations from an application's class hierarchy. For our running example, the flattened class hierarchy, in which classes are siblings rather than subtypes and supertypes, is shown in Figure 6.3. The corresponding code after flattening is shown in Figure 6.2(a).

To preserve the semantics of the application, instance fields and methods in the original program are copied from classes to their subclasses, and interface types are created to offer a common interface that would otherwise be provided through inheritance. The interface types are used throughout the application's code whenever possible; the types of variables, fields, and method parameters are replaced by their corresponding interface types to get rid of much of the original type information.

Before instance fields and methods are copied, the fields are first encapsulated to make them accessible through the interface types. In our example, methods getB/setB and getS/setS are added to X1 and Y2, respectively. During the copying, methods and fields are renamed as needed to avoid collisions with existing ones. Constructors, which cannot be renamed, are given an extra, distinguishing parameter. Examples include method m, which is renamed to m1 after being copied from X1 to X3, and the constructor with parameter type list [A,B], which is given an extra argument of type D after being copied from X1 to X2.

Each interface created during flattening declares all methods defined in the classes that implement the interface. This is done to ensure that after the declared type of variables is changed to one of the interface types,



(a)

```
class X3 extends X2{
 X3(A a, C c, short s){
  super(a, c);
  X2 x2 = new X2(a, c);
  x2.n();
  Y2 y2 = new Y2(s, c);
  y2.p(x2);
  y2.p(y2);
 }
 void m(){
  X1 x1 = new X1(null, b);
  x1.m();
  super.m();
 }
}
```

(b)

Figure 6.1: Example program. (a) Original class hierarchy. (b) Original code.

```
class X3 implements I1{
 X3(A a, C c, short s){
  this(a, c);
  I1 x2 = new X2(a, c);
  x2.n();
  I2 y2 = new Y2(s, c);
  y2.p(x2);
  y2.p(y2);
 }
 void m(){
  I1 x1 = new X1(null, b);
  x1.m();
  m1();
 }
}
```

(a)

```
class X3 implements I{
 X3(A a, C c, short s){
  this(a, c);
  I x2 = IFactory.create(a, _, c, _, _, ...);
  x2.mrg1();
  I y2 = IFactory.create(_, _, c, _, s, ...);
  y2.mrg4(x2, _);
  y2.mrg2(y2);
 }
 void m(){
  I x1 = IFactory.create(null, _, c, _, _, ...);
  x1.m();
  mrg3(_);
 }
}
```

(b)

Figure 6.2: Code of Figure 6.1(b) after transformation. (a) Code after CHF. (b) Code after CHF, IM, MM, and OFI. In both figures copied fields and methods, and dummy methods have been omitted for brevity. In (b) underscores represent arguments that can be chosen arbitrarily.

methods can still be invoked on the objects bound to those variables. Interface I1, for example, declares method m such that m can still be invoked on x1 after the declared type of x1 is changed to I1. Furthermore, to ensure that each class defines all the methods in its governing interface,

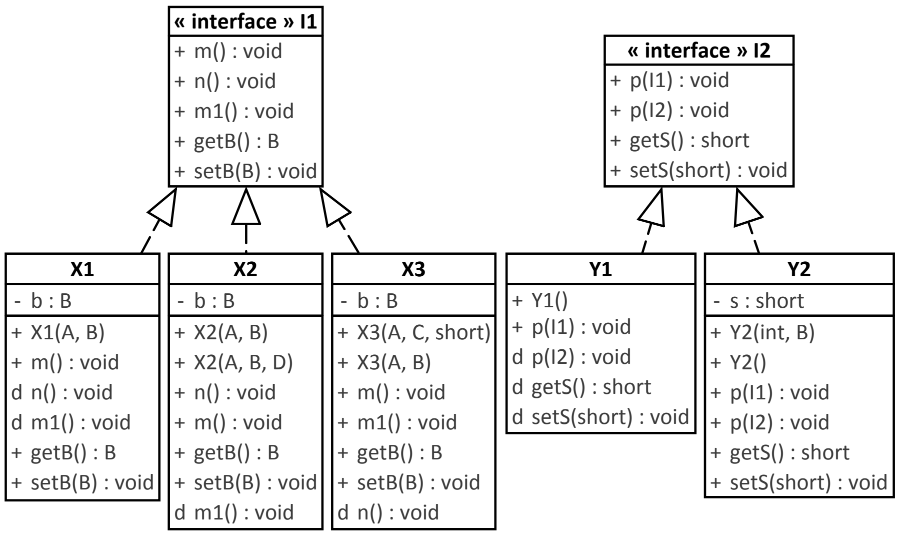


Figure 6.3: Class hierarchy of Figure 6.1(a) after CHF.

dummy methods are added. In Figure 6.3, the dummy methods are shown with a letter ′d′ instead of their visibility modifier.

6.1.2 Interface Merging and Method Merging

As discussed previously, flattening the class hierarchy is in many cases not sufficient to remove most of the type information. From the class hierarchy in Figure 6.3, for instance, an attacker can still deduce that classes X1, X2, and X3 were related in the original hierarchy, because they implement the same interface. Furthermore, since each interface is implemented by a (relatively) small number of classes, much type information can still be deduced from the assignments and method signatures in Figure 6.2(a). To remove this information, we proposed interface merging. This technique combines multiple interfaces created during flattening, resulting in fewer interfaces that are implemented by more (unrelated) classes. However, an important side-effect of this technique is that, as the total number of interfaces decreases, the number of methods per interface increases. Hence, additional dummy methods need to be added to the implementing classes to ensure that each class defines all methods declared in its corresponding merged interface.

In practice, the number of dummy methods can increase quickly, even for small programs with only a few interfaces. As shown in Figure 6.4, merging interfaces I1 and I2 from Figure 6.3 increases the number of dummy methods from 7 to 29. This can be problematic, as dummy methods can contribute significantly to the size of an application, even when their bodies are empty[1]. To remove them, we presented method merging in Chapter 3. This technique iteratively and greedily selects sets of same-signature methods to merge. It operates on sets of same-signature methods rather than on individual methods to keep the relationship between overridden and overriding methods intact. In each iteration of the algorithm, a pair of method sets (S_i, S_j) is selected for merging. Selection happens based on the number of changes that will have to be made to the signatures of the methods when the sets are merged. Pairs that result in few changes are chosen first, while those that result in many changes will only be chosen in later iterations. During merging, each pair of methods $m_i \in S_i$ and $m_j \in S_j$ declared in the same class is replaced by a new method m_{ij} whose parameter type list and return type consist of the merged parameter type lists and return types of m_i and m_j. To avoid having to merge to two non-empty method bodies, method merging requires that each pair of methods contains at least one dummy method. The algorithm terminates when a pair of method sets for which this holds can no longer be found.

Figure 6.5 shows the class hierarchy after method merging. In this hierarchy, void n() and short getS() have been combined into short g1(), B getB() and void p1(I) into B g2(I), void m1() and void setS(short) into void g3(short), and void p(I) and void setB(B) into void g4(I,B). At this point, there are only 11 dummy methods left, which cannot be further be merged into other methods. Changes to the code as a result of method merging are shown on lines 5, 7, 8, and 13 of Figure 6.2(b). On lines 7 and 13 additional arguments are provided because g4 and g3 require more arguments than p and m1, respectively.

6.1.3 Object Factory Insertion

After CHF, IM, and MM, some type information can still be deduced from object creation sites. The object factory insertion transformation tries to remove it by replacing constructor calls by calls to obfuscated object factories. For maximum effect, each factory is constructed such that it can create objects of as many different types as possible. This

[1]Note the large differences in application size overhead between program versions obtained with and without method merging, as shown in Figure 5.4(a).

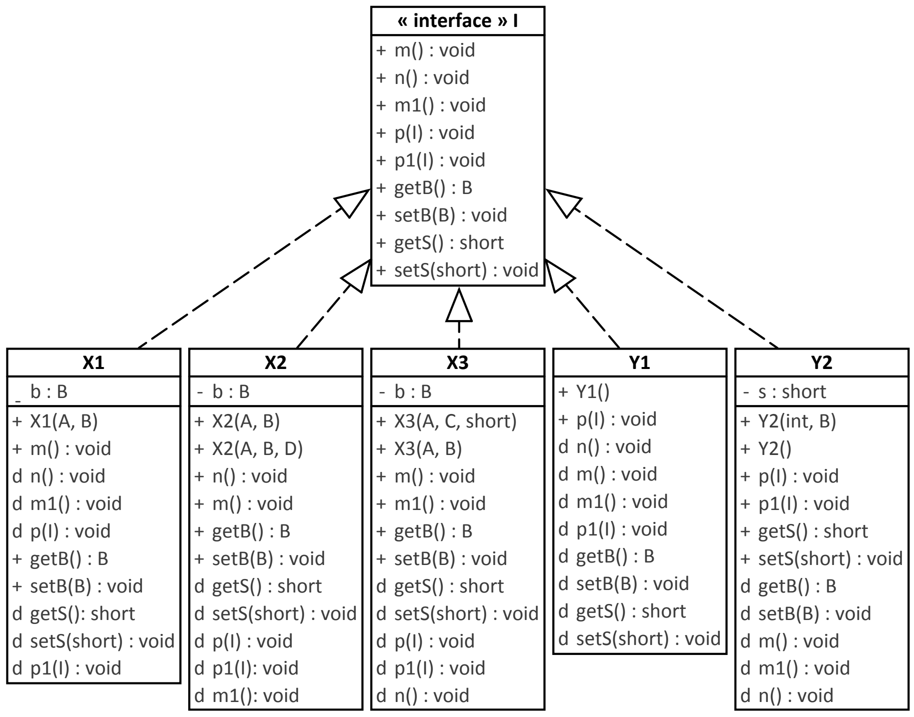


Figure 6.4: Class hierarchy after CHF and IM.

makes it more difficult to narrow down the exact type of object that will be created at different program points.

Applying OFI to the running example results in the single object factory shown in Figure 6.6. It has a single create method that can return instances of all five classes in the application. To enable this, the factory method's parameter type list consists of a combination of the parameter type lists of the classes' constructors. This is done to ensure that arguments can be provided for all eight constructors, any of which may be called. Selecting which constructor to invoke is done by choosing the appropriate values for a number of additional parameters, indicated by the dots on line 2. For instance, to replace the object creation on line 6 of Figure 6.2(a), the values of these parameters should be chosen such that the expression in the if-statement on line 9 of Figure 6.6 evaluates to true. The values for factory method parameters that are not passed on to the selected constructor can be chosen arbitrarily, as indicated by the underscores on line 6 of Figure 6.2(b).

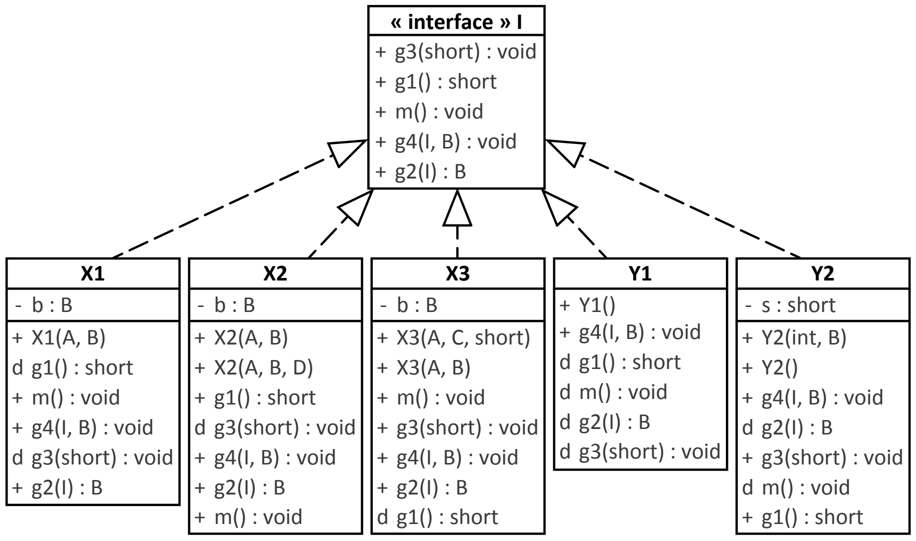


Figure 6.5: Class hierarchy after CHF, IM, and MM.

6.1.4 Causes of Overhead

For small applications like our running example, the overhead introduced by CHF, IM, and OFI is minimal. However, as we experienced during our evaluation in Chapter 5, the overhead can be very large for realistic applications. For heavily obfuscated versions of the eclipse benchmark, we measured increases in application size[2] and execution time of over six and eight times, respectively. The two main causes of overhead were found to be method merging and object factory creation.

Effect of Method Merging

Method merging is based on the idea that removing dummy methods by merging them into other methods will reduce application size. While this generally holds during the first iterations of the method merging algorithm presented in Chapter 3, it does not always continue to hold as

[2]In the remainder of this dissertation we use application size rather than code size to denote the size of an application. This is because Java applications consist of more than just code. They also contain much meta-data that is stored in constant pools and attributes. During transformation, this meta-data also needs to be updated, and the size of the application may change as a result of those updates.

```
class IFactory {
 static I create(A a, B b, C c, D d, int i, short s, …) {
  if(…) return new X1(a, b);
  if(…) return new X2(a, b);
  if(…) return new X2(a, b, d);
  if(…) return new X3(a, c, s);
  if(…) return new X3(a, b);
  if(…) return new Y1();
  if(…) return new Y2(i, b);
  return new Y2();
 }
}
```

Figure 6.6: Factory class created by OFI.

more and more methods are merged. This is because method merging only reduces the size of an application as long as the space saved by removing dummy methods is larger than the extra space needed to store (i) the descriptors[3] of the merged methods, and (ii) the code required to push dummy arguments onto the stack. However, because the existing algorithm is unbounded, sets of methods are simply merged until a pair of mergeable sets can no longer be found. As a result, the methods' parameter type lists and descriptors can grow so large that method merging actually increases the size of the application. For the eclipse benchmark, for example, method descriptors can make up 68% of the total size of the application, as a result of excessive method merging.

To avoid such cases, one could introduce an upper bound on the length of method descriptors. However, this coarse-grained approach does not solve the underlying problem and still leads to results that are far from optimal. This is because choosing an upper bound that is too low may result in missed opportunities for reducing an application's size, while choosing a larger one may lead to unwanted increases in the application's size. Furthermore, we expect the optimal upper bound to be application-dependent, which is not desirable. Ideally, we should have a metric that determines for each merge operation whether or not it will lead to a decrease in application size, and that can be computed efficiently and used effectively during method merging.

An additional problem with the existing method merging algorithm is that method sets are merged suboptimally. During merging, the names and parameter type lists of merged methods are randomized to avoid

[3]Human-readable strings consisting of a method's return type and parameter types.

collisions with existing methods. As a result, many methods have unique signatures. From an application size perspective, this is very inefficient, because a separate name and descriptor has to be stored for each method. It is much more space-efficient to have multiple methods with similar signatures, such that their names and/or descriptors can be shared.

Hence, what we need is a method merging algorithm that stops whenever merging is no longer beneficial, and that reduces the variation in method signatures to avoid excessive application size overhead in the form of constant pool entries that represent unique method names and descriptors. In Section 6.2 we present such an algorithm.

first reference ever to constant pool, make sure it is introduced before. introduction seems like the place

So far we have only focused on application size. However, long parameter type lists generated during method merging also impact the performance of applications, since they imply that more arguments need to be passed for each method invocation. In this dissertation we do not provide a targeted solution to this problem. Instead, we show by means of experimentation that our new method merging algorithm automatically results in less performance overhead, because it generates parameter type lists that are much shorter on average.

Effect of Object Factory Insertion

Our evaluation in Chapter 5 revealed that object factory insertion is the most expensive transformation in terms of execution time overhead, accounting for 393 of out 670 percentage points of the measured slowdown in extreme cases. The main cause of this overhead are the many additional arguments that need to be passed to the factory methods, compared to when the constructors are invoked directly. Even for our running example already more than six additional arguments are required to create an instance of class Y1 using the factory in Figure 6.6. For realistic applications, this number can become much larger. For instance, the factory methods of heavily obfuscated versions of the eclipse benchmark require on average 32 arguments more than the individual constructors. In extreme cases, the difference can even be as large as 70.

The performance overheads observed for OFI are hence not surprising, especially since the algorithm makes no attempt at reducing the cost of the factories. Instead, it just tries to confuse static analyses maximally by constructing factory methods that invoke as many constructors as possible. While this is desirable from a type obfuscation point of view, it can also result in much overhead. The main problem with OFI as presented in Chapter 4, is that each factory class only contains a single factory method that invokes all possible constructors. As a result, each factory method needs to be able to provide arguments for many different constructors that often require a different number of arguments and/or arguments of different types. In many cases, combining all these constructors' parameter type lists results in parameter type lists for the factory methods that are much larger than those of the constructors.

So far, we have only discussed the reasons why the parameter type lists of the factory methods are so large. However, the overhead of OFI is not solely determined by the number of arguments of the factory methods. The number of times each factory method is invoked is equally important. This is because the overhead of a factory method will only truly be noticeable when it is invoked frequently.

Based on the above observations, we developed an algorithm that tries to reduce the number of factory method parameters by limiting the variation in their types. It uses profile information to construct factory methods intelligently based on how many times each constructor is invoked and on how many additional arguments are required when calls to the constructors are replaced by calls to factory methods. An overview of the algorithm is given in Section 6.4.

6.2 Cost-effective Method Merging

Based on the observations made in Section 6.1.4, we developed a bounded method merging algorithm that iteratively and greedily merges method sets that result in the largest decrease in application size. It works as follows.

- Let $\mathbb{I}$ be the set of all subtree interfaces after class hierarchy flattening or interface merging, $\mathbb{M}$ the set of all methods, $\mathbb{C}$ the set of all classes, $\mathbb{T}$ the set of all types, and $\mathbb{N}$ the set of all natural numbers.
- Let N_k be a set of possible method names that can be stored in k bytes or less.
- Let $M : \mathbb{T} \mapsto \mathbb{M}$, with $M(t)$ the set of all methods declared in t.

Algorithm 6.1: Cost-effective method merging.

```
$\mathfrak{S} = \{\mathfrak{s} = (S_1, S_2, n, p, r, g) \in \mathbb{S} \times \mathbb{S} \times N_k \times \mathbb{T}^* \times \mathbb{T} \times \mathbb{N} \mid \text{valid}(\mathfrak{s})\}$
while $\mathfrak{S} \neq \emptyset$ do
    $(S_1, S_2, n, p, r, g) = \arg\max_{\mathfrak{s} \in \mathfrak{S}} \mathfrak{s}.g$
    $S = \text{merge}(S_1, S_2, n, p, r)$
    $\mathbb{S} = \mathbb{S} \cup \{S\} \setminus \{S_1, S_2\}$
    $\text{update}(\mathfrak{S}, S_1, S_2, n, p, r, S)$
```

- Let $S : \mathbb{M} \mapsto \mathcal{P}(\mathbb{M})$ be the function for which $S(m)$ is the set of methods that at all times should have the same signature as m.
- Let $f : \mathbb{M} \mapsto \{false, true\}$, with $f(m)$ indicating whether the signature of m can be changed.
- Let $\mathbb{S} = \{S(m) \mid \forall\, i \in \mathbb{I}\, \forall\, m \in M(i)\, \forall\, n \in S(m).\, f(n)\}$ be the set of all method sets that can be merged.
- Let $N : \mathbb{C} \times \mathbb{S}^2 \mapsto \mathbb{M}$, with $N(c, S_i, S_j)$ the methods of class c that will be merged when S_i and S_j are merged, i.e.,

$$N(c, S_i, S_j) = M(c) \cap (S_i \cup S_j).$$

The algorithm can now be written as shown in Algorithm 6.1. Each tuple (S_1, S_2, n, p, r, g) in $\mathfrak{S}$ corresponds to a potential merge operation and consists of the two method sets S_1 and S_2 that will be merged, as well as the name n, parameter type list p, and return type r of the methods that will be created when S_1 and S_2 are merged. Each tuple also contains the gain g in number of bytes as a result of performing the merge operation.

In the following sections we explain the subroutines valid, merge, and update used by Algorithm 6.1 in more detail. First, we state the conditions that must hold for a tuple to be considered valid. Then, we discuss how method sets are merged. Next, we explain how the set $\mathfrak{S}$ is updated after each merge operation. Finally, in Section 6.2.4 we discuss how the gain of each tuple is computed.

6.2.1 Valid Merge Operations

The set $\mathfrak{S}$ only contains valid tuples. A tuple (S_1, S_2, n, p, r, g) is valid if, and only if the following conditions hold:

C.1 p is a normalized version of the merged parameter type lists of the methods in S_1 and S_2.

C.2 All methods in S_1 and S_2 either have the same return type r, or for one set the return type is void, and for the other it is r.

C.3 No class has a reaching non-dummy implementation for two or more methods from $S_1 \cup S_2$.

C.4 Each class c for which $N(c, S_1, S_2)$ is not empty

(a) does not declare a method m with signature $<r, n, p>$, unless $m \in N(c, S_1, S_2)$;

(b) does not have a reaching implementation for a method with signature $<r, n, p>$, unless $N(c, S_1, S_2)$ contains a method with that signature, or $N(c, S_1, S_2)$ only contains dummy methods.

C.5 There is no tuple $(S_1, S_2, n_m, p, r, g_m)$, with $n_m \neq n$ and $g_m > g$ for which C.4 also holds.

As defined in Section 3.4, a class is said to have a reaching implementation for a method if that class or one of its superclasses provides an implementation for that method.

Conditions 2 and 3 express the merge condition of the original method merging algorithm presented in Chapter 3. They ensure that two non-dummy methods are never merged, that the methods in S_1 and S_2 will not erroneously override each other after merging, and that their return types are compatible.

Condition 4 prevents merged methods from incorrectly replacing or overriding methods that are not being merged. In the original algorithm there was no need to check for this, as potential collisions were trivially avoided by assigning unique random names to the merged methods. However, because the new algorithm tries to reduce application size by decreasing the variation in method names and descriptors, collisions are likely and must be avoided.

It may seem counter-intuitive that in one case, condition 4 (b) does allow merged methods to override other methods. However, this only happens if the set $N(c, S_1, S_2)$ only contains dummy methods. In that case the merged method will also be a dummy method, and because it overrides another method, it can (and must be) removed from the program. This happens in the merge subroutine.

Conditions 1 and 5 ensure that the algorithm effectively reduces application size by ensuring that the names and the descriptors of the merged methods are chosen optimally. Condition 1 requires that the merged parameter type lists are normalized. This increases the likelihood that multiple methods have the same descriptor, and that fewer constant pool entries are needed. The original algorithm does not have this requirement. Hence, when merging the parameter type lists [A, B] and [C], it can generate any of the six different permutations of [A, B, C], which each require a separate constant pool entry. The new algorithm, by contrast, will always merge these parameter type lists into their canonical form [A, B, C]. It will never generate any of the other permutations.

Finally, condition 5 states that the names of merged methods should be chosen greedily to maximize the decrease in application size. If there are multiple names that result in the same maximum gain, one of them is chosen at random.

6.2.2 Merging Method Sets

Given a valid tuple (S_1, S_2, n, p, r, g), the merge subroutine on line 4 of Algorithm 6.1 performs the following steps:

1. Create an empty set S to hold the merged methods.
2. For each class c for which the set N_c of all methods to merge, with $N_c = N(c, S_1, S_2)$, is not empty:
   (a) Create a new method m with name n, parameter type list p, and return type r.
   (b) Make the body of m 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) Remove all methods in N_c from c.
   (d) Add m to c and to S.
3. Rewrite invocations of methods in $S_1 \cup S_2$ to invocations of their corresponding methods in S, adding dummy arguments as needed.
4. Remove from S and from their declaring classes all dummy methods that override methods.
5. Return S.

Step 4 handles the case where condition 4 (b) allows dummy methods to override regular methods.

6.2.3 Updating the Set of Valid Merge Operations

Each time a merge operation is performed, the program being transformed changes, as methods are added and removed, and signatures are rewritten. As a result, some previously considered merge operations may no longer be valid, and others may become available. The goal of the update subroutine on line 6 of Algorithm 6.1 is to ensure that after merging, $\mathfrak{S}$ again contains all valid merge operations. It performs the following steps:

1. Remove from $\mathfrak{S}$ all tuples involving S_1 or S_2.
2. For each tuple $(S_{1i}, S_{2i}, n_i, p_i, r_i, g_i) \in \mathfrak{S}$ for which (i) condition 4 does not hold and $<r_i, n_i, p_i>$ equals $<r, n, p>$, or (ii) condition 5 does not hold and $\hat{C}(S_{1i}, S_{2i}) \cap \hat{C}(S_1, S_2) \neq \emptyset$:
   (a) remove $(S_{1i}, S_{2i}, n_i, p_i, r_i, g_i)$ from $\mathfrak{S}$;
   (b) find a name $n_j \in N_k$ such that conditions 4 and 5 hold for $\mathfrak{s}_j = (S_{1i}, S_{2i}, n_j, p_i, r_i, g_j)$. If such an n_j exists, add $\mathfrak{s}_j$ to $\mathfrak{S}$.
3. Create new valid merge operations involving S, if they exist, and add them to $\mathfrak{S}$.

In this algorithm, $\hat{C}$ is defined as $\hat{C} : \mathbb{S}^2 \mapsto \mathcal{P}(\mathbb{C})$, with $\hat{C}(S_i, S_j)$ the set of all classes whose size may change as a result of merging S_i and S_j. This set includes all classes that declare and/or refer to any of the methods in S_i or S_j, because these classes have constant pool entries for those methods.

6.2.4 Computing the Change in Application Size

In order to decide whether a merge operation is beneficial and a corresponding tuple should be added to $\mathfrak{S}$, the algorithm needs to determine the change in application size the merge operation will induce. In other words, it needs to compute the gain g for that tuple. For Java applications, this comes down to computing the change in size of all class files affected by the merge operation. Unfortunately, this is not straightforward, because program transformation tools such as obfuscators generally do not operate directly on class files. Instead, obfuscators are often built on top of tools such as ASM [17], BCEL [27], and Soot [74] that use an intermediate representation of the class files to allow for easier analysis and transformation. Because these representations hide many details of the underlying class file format, accurately determining the (change in) size of an application from its intermediate representation is difficult.

The easiest way to circumvent this problem is to convert the program's intermediate representation (IR) to class files, and do the calculations directly on the class files. However, this approach is extremely slow[4]. In order to compute the effect a merge operation will have on the application's size, the merge operation needs to be performed on the program's IR, class files need to be generated from the IR, and the changes to the IR need to be undone such that the effect of the next operation can be computed, or a merge operation can be performed. For heavily obfuscated versions of some large benchmarks there are easily tens of millions of potential merge operations. To compute the gains for all of these, billions of class files would need to be generated, which makes this approach impractical.

As a better solution to this problem, we developed a model for application size. This model maps each class' intermediate representation to a light-weight representation of the class file that is generated when the class is written to disk. During merging, the model is updated to reflect the changes to the application as a result of merge operations. By keeping the model up to date, the class file representations do not need to be regenerated each time application size calculations need to be made. Furthermore, rather than trying out merge operations on the program's IR, they can be performed on the model. Afterwards, the state of the model can be reset easily to compute the effect of another merge operation. We give an overview of our model in the next section.

6.3 A Model for Application Size

For (S_1, S_2, n, p, r, g) to be a valid tuple, g must be equal to the gain in application size after calling $\mathrm{merge}(S_1, S_2, n, p, r)$. To compute g the method merging algorithm therefore needs to compute how each of the individual steps of the merge subroutine affect the size of the application.

[4]For some benchmarks it takes on average 15 ms per class to convert the class' IR into an in-memory class file whose size can be computed. For those applications it can take multiple days to generate a single transformed program version.

In doing this, it makes a distinction between changes in size as a result of meta-data changes, and those as a result of changes to the method bodies. The latter are easy to compute based on the size of the instructions that are added or removed. Tracking changes in the size of the meta-data is more complex, because the meta-data in class files is stored as a directed graph. Whenever a node is removed from this graph, other nodes may also no longer be needed, depending on whether they are still referenced or not. Hence, without proper knowledge of the references between different parts of the meta-data, the algorithm cannot know whether or not removing a certain method will also allow it to remove the constant pool entries that hold its name and descriptor. As a result, it also cannot compute the change in meta-data size.

To enable the algorithm to track changes in the meta-data, and compute changes in its size as a result of method merging, we developed meta-data graphs.

6.3.1 Meta-data Graphs

The meta-data graph (MDG) of a Java class is a simplified graph representation of that class' underlying class file. In Java, each class file contains the definition of a single class or interface. This definition includes the name of the class, the names of its superclass and interfaces, and a description of the class' fields, methods, code, and attributes, as well as a description of other classes, fields, and methods referenced from the class' code. In a class file, this information is organized in a variable-length ClassFile structure defined in the Java Virtual Machine Specification [48]. This structure is essentially a serialized representation of a directed graph of (variable-length) structures in which the root represents the class itself and every other node represents a constant pool entry, a field, a method, or an attribute.

Meta-data graphs are simplified versions of those graphs. They also contain a separate node for each instance of each of the structures in a class file. However, each node only contains its type and its size, no actual data. The size of each node is computed as the size of the corresponding structure, as defined in the Java Virtual Machine Specification [48]. There is one exception: since we only use the graphs to model changes in the size of meta-data, we assume that instructions occupy no space.

Figure 6.7 shows partial MDGs[5] for two classes A and B. Class A has

[5]For clarity, we omitted many of the nodes. In practice, meta-data graphs can easily

three methods: void a(), void b(), and void c(int) that do not invoke any other methods. Class B has a single method void b() that calls method void b() defined in class A. The MDG for class A contains one node corresponding to the ClassFile structure that represents the class. This node refers to a node of type UTF8_info[6] that represents the class name. The ClassFile node also refers to three method_info nodes, one for each of A's methods. Each method_info node further refers to UTF8_info nodes for the name and the descriptor of the method. Additionally, each method_info node has an associated Code_attribute node that represents the method body. In our case, each Code_attribute has the same size, since we do not take into account the size of instructions, only the size of the meta-data. For class B, there is an edge from the Code_attribute node for method void b() to a Methodref_info node that represents method void b() defined in class A. The Methodref_info refers to two additional nodes: a Class_info node that represents class A, and a NameAndType_info node that represents the signature of the invoked method. From this node, there are two more edges to UTF8_info nodes that represent the name and the descriptor of the invoked method, respectively.

Computing the gain in meta-data size for a class using its MDG is straightforward. Whenever an edge in the graph is removed, and a node can no longer be reached from the root node as a result, the gain is increased by the size of that node. Similarly, when an edge is added and a node becomes reachable, the gain is reduced by the size of that node. Determining the gain in a program's meta-data size as a result of a merge operation is hence just a matter of performing the right operations on the MDGs of its classes.

6.3.2 Modeling High-level Changes

The merge subroutine of Algorithm 6.1 performs three tasks that can affect the size of an application: removing methods, adding methods, and updating method invocations. They are performed in steps 2 (c) and 4, 2 (d), and 3, respectively, as described in Section 6.2.2.

For efficiency reasons, we do not model method removal and method addition for the exact methods described in the algorithm. Instead, we only model method removal for dummy methods. For all other methods,

contain hundreds to thousands of nodes of over thirty different types.

[6]In the Java Virtual Machine Specification, all structures that define constant pool entries have a name that starts with CONSTANT_. For brevity, we dropped this prefix.

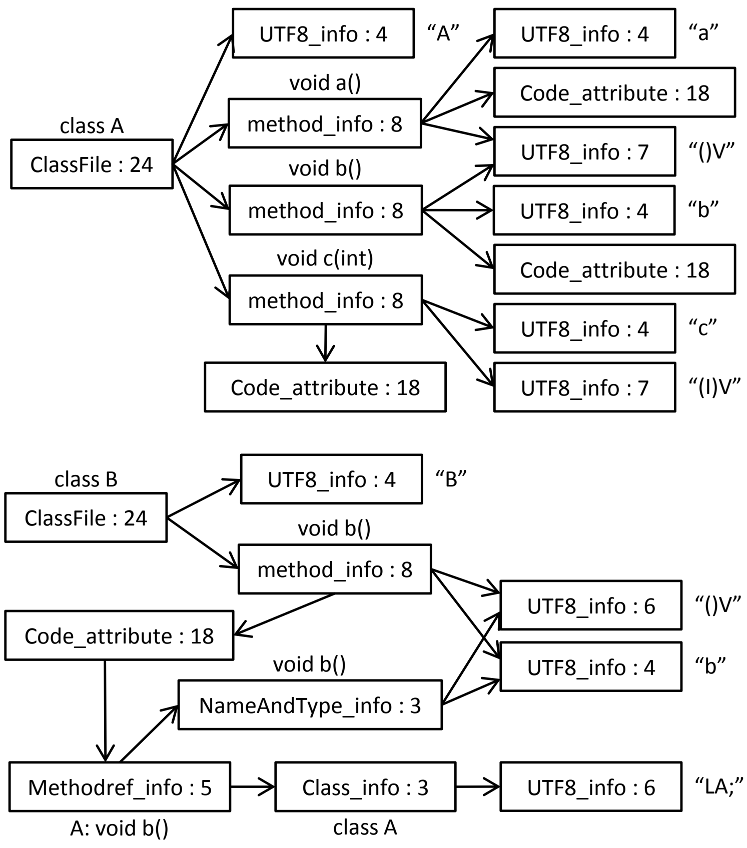


Figure 6.7: Partial meta-data graphs for classes A and B.

we model signature changes. The reason is that during merging, non-dummy methods will never be removed permanently from the program. They are removed temporarily in step 2 (c) of the merge subroutine, only to be added again (with a possibly different signature) as one of the merged methods in step 2 (d). Hence, if we would model method removal and addition separately, we would first deduct the size of the methods' instructions and their meta-data, only to add it back when the merged method is added.

By avoiding this, our model is not only more efficient, but also much simpler, because it does not have to compute changes in bytecode size for arbitrary methods. In fact, as we will see in the next sections, it only has to compute changes in size as a result of changes to high-level return and call instructions. This is an important advantage, because some program transformation frameworks use virtual register-based intermediate representations that do not directly map to the stack-based bytecode [12, 74]. In such cases, computing the size of the bytecode instructions from the high-level statements can be hard without at least (partially) simulating the conversion of these statements to bytecode.

To summarize, our model has support for three operations: removing dummy methods, updating method signatures, and updating method invocations. In the next sections, we explain how these operations are modeled, and we show how the gain in application size for our running example is computed when methods void b() and void c(int) of class A are merged into void a(int). For demonstration purposes, we assume that method c is a dummy method, and method b is a non-dummy method.

Removing Dummy Methods

Each time a dummy method would be removed from a class in step 2 (c) or step 4 of the merge subroutine of Algorithm 6.1, the edge from that class' ClassFile node to the method's method_info node needs to be removed. In our case, removing dummy method void c(int) would reduce the application's size by 37 bytes, because after the edge from class A's ClassFile node to c's method_info node is removed, the method's Code_-attribute node and the nodes representing its name and descriptor are no longer reachable.

Considering that we previously stated that the bodies of dummy methods are empty, one would expect that removing a dummy method has no impact on the size of the code. However, this is not true, since it is not allowed for a method to have a truly empty body. At the very least, each method's body should contain a return instruction that returns a value that is compatible with the return type of the method. For void methods, a simple return instruction can be encoded in one byte, while for non-void methods returning zero or null requires two bytes. Since c is a void method, only one additional byte is saved by removing it. This brings the total gain for this step to 38 bytes.

Updating Method Signatures

For methods whose bodies would be selected in step 2 (b) of the merge subroutine, we model signature changes. For each such method m of which the signature differs from the new signature $<r, n, p>$ the algorithm does the following.

1. If the name of m is equal to n, do nothing. Otherwise, delete the edge from m's method_info node to the UTF8_info node representing its name. Then, add an edge from m's method_info node to the UTF8_info node for n. If such a node does not exist, create it first.
2. Repeat step 1 for the method descriptor.

For the running example, the signature of void b() is changed to void a(int). Because both the name and the descriptor of b need to be updated, we first remove the edge from b's method_info node to the UTF8_info nodes for "b" and "()V". As a result, the node for the string "b" becomes unreachable, and we gain 4 bytes in application size.

Next, we add edges from b's method_info node to the nodes corresponding to the new name and descriptor, which are "a" and "(I)V", respectively. This increases the size of the application by 7 bytes, because the node for "(I)V" has become reachable again. The node for "a" was already reachable from the method_info node for a, so adding an additional edge to it does not affect the size of the application. The net result of this step would hence be an increase in application size of 3 bytes, or a gain of -3 bytes.

Updating Method Invocations

After the signature of a method is updated, changes need to be made to the instructions that call that method. Furthermore, if the number of arguments of the method has increased, additional instructions need to be added that push (fake) arguments onto the stack. For each method m whose signature is updated in the previous step, we do the following.

For each class c that has a method that calls m, perform the following operations on its MDG:

1. Let m_{ref} be the Methodref_info node describing m. Remove the edge from m_{ref} to the NameAndType_info node representing the old signature of m.
2. Create UTF8_info nodes for m's new name and descriptor, if these do not exist.
3. Add an edge from m_{ref} to the NameAndType_info node representing m's new signature <r, n, p>. If the latter does not exist, create it and add edges from it to the UTF8_info nodes representing m's new name and descriptor.

Also, for each invocation in any of c's methods, decrease the gain in application size estimated for this merge operation by the difference in length between m's old and new parameter type lists. This step models the fact that for each extra parameter, one additional byte is required to encode an instruction that pushes either zero, or null onto the stack.

Since method void b() of class B calls void b() in class A, we need to update the MDG of class B. More specifically, we need to remove the edge from A: void b()'s Methodref_info node to the NameAndType_info node for void b(). This increases the gain by 3 bytes. Next, we create UTF8_info nodes for the name "a" and the descriptor "(I)V", because these do not yet exist. Finally, we create a new NameAndType_info node for the new signature A: void a(int), and add an edge from the Methodref_info node to that node, and an edge from that node to the nodes for "a" and "(I)V". Adding these edges reduces the gain by 3, 4, and 7 bytes, respectively.

Assuming that B: void b() contains only one call to A: void b(), only one extra push instruction is required when the signature of A: void b() is changed to A: void a(int). As a result, the size of application increases by 1 byte. If we now add up the total gains for each of the steps, we get that merging void b() and void c(int) of class A into void a(int) results in a total gain in application size of 23 bytes.

6.3.3 Undoing Changes

In between computing gains in application size for different merge operations, the model needs to be reset. This is where the model really shows its strength, being much simpler and faster to reset than the program's intermediate representation. One of the reasons why the model is faster is that it never changes any of the program's instructions. As a result, our algorithm does not need to restore the method bodies, only the meta-data. Furthermore, since changes to the meta-data are modeled using MDGs, restoring them is just a matter of undoing the operations on those graphs. To do this, the algorithm simply needs to keep track of which nodes and edges have been added and removed while performing the operations discussed in the previous sections. After each computation, it can then simply remove newly created edges and nodes, and restore the old edges. Nodes do not need to be restored, as they are never removed; they only become unreachable. As a result, the code that undoes the operations on the MDGs is fast and simple. It just needs to iterate over a list of nodes and edges and add or remove them as necessary. By not having to invoke any high level APIs to restore the program's IR, we avoid potential overhead from extra objects that need to be created, or caches that need to be updated.

If one of the merge operations is selected and actually performed on the program's intermediate representation, the model is updated alongside it, just like it would when computing the gain. The main difference is, of course, that the model is not reset afterwards.

6.3.4 Limitations

In some cases, the gain computed by the application size model differs from the actual gain. Because the model only computes changes in instruction size as a result of return instructions that are removed, or invoke instructions that are altered, it assumes that the size of all other instructions remains the same. However, in practice this is not always true. A program transformation framework may decide to generate different instructions, depending on the specific index of the constant pool entry an instruction refers to. For instance, if the index of a constant that needs to be pushed onto the stack can fit in a single byte, tools may choose to generate an ldc instruction, instead of the larger ldc_w instruction which encodes two-byte indices.

A similar problem occurs with bytecode offsets. As more instructions are generated to push arguments onto the stack, the body of a method may become so large that offsets can no longer be encoded in two bytes. As a result, the framework may need to generate additional code to implement trampolines, or use wide versions of the goto and jsr instructions, which encode 4 byte offsets.

In practice, the error is small. For instance, in all our experiments, we have never encountered cases where bytecode offsets became so large that additional trampoline code had to be inserted, or that goto_w or jsr_w instructions needed to be generated. We have only encountered single-byte errors as a result of ldc instructions begin generated instead of ldc_w, or vice-versa. These errors are mostly negligible, since the indices of constant pool entries do not change drastically during method merging, and because ldc and ldc_w only account for 3% of all instructions.

6.4 Cost-effective Object Factories

As discussed in Section 6.1.4, the overhead of OFI increases with each additional argument required by the factory methods, and with each object created using those methods. Unfortunately, we cannot easily reduce the number of objects created during the execution of an application without altering the application's semantics. We therefore opted to lower the overhead by limiting the number of arguments of each factory method based on how many times each of the constructors it invokes need to be executed. To achieve this, we developed a new profile-driven algorithm for creating object factories. Contrary to the algorithm from Chapter 4, which generates a single factory method for each set of constructors, the new algorithm may generate multiple factory methods that each call a subset of the constructors, or one large factory method that invokes all of them. The main benefit of having multiple factory methods is, of course, that each factory method requires fewer arguments, as they each have to provide arguments for a smaller set of constructors.

Another important difference compared to the algorithm presented earlier, is that the new algorithm initially treats each constructor as if it only accepts arguments of type java.lang.Object, and of the primitive types int, long, float, and double. All other types are mapped to these five. In doing this, it significantly reduces the number of different types that can occur in the parameter type lists of the constructors, which results in factory methods that require far fewer arguments. Of course, when the constructors are actually invoked, arguments of the correct types need to be passed to them. In that case, the generic arguments passed to the factory methods are cast to the types expected by the constructors.

As an example, Figure 6.8 shows an object factory for the application in Figure 6.1(b), obtained using our new algorithm. Because the algorithm treats the parameter type lists of the eight constructors of X1, X2, X3, Y1, and Y2 as [Object, Object], [Object, Object], [Object, Object, Object], [Object, Object, int], [Object, Object], [], and [int, Object], and [], respectively, they can be combined into [Object, Object, Object, int] instead of [A, B, C, D, int, short]. As a result, the new factory method requires two arguments less than the one shown in Figure 6.6. Lines 3-7 and 9 of Figure 6.8 include the type casts needed to cast the arguments of the factory method, which are passed as instances of type java.lang.Object and values of type int, to the types A, B, C, D, and short that are expected.

The factory shown in Figure 6.8 is just one of the possible factories the new algorithm can generate. If, based on the profile information, the algorithm decides that generating two different factory methods will result in less overhead, it may generate the factory shown in Figure 6.9. In this case, calls to the original constructors are split between two different

```
class IFactory {
 static I create(Object o1, Object o2, Object o3, int i, ...) {
  if(...) return new X1((A)o1, (B)o2);
  if(...) return new X2((A)o1, (B)o2);
  if(...) return new X2((A)o1, (B)o2, (D)o3);
  if(...) return new X3((A)o1, (C)o2, (short)i);
  if(...) return new X3((A)o1, (B)o2);
  if(...) return new Y1();
  if(...) return new Y2(i, (B)o1);
  return new Y2();
 }
}
```

Figure 6.8: New object factory with a single factory method.

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

Figure 6.9: New object factory with two factory methods.

factory methods. Also, an additional call to a newly created no-argument constructor for class X1 is added on line 10 to ensure that each factory method invokes at least one constructor of each class.

To construct factories such as the ones in Figures 6.8 and 6.9, our algorithm operates in three steps. In the first step, information about object creation sites in the program is gathered to determine the properties of the object factories. While collecting this information, each constructor is modeled as a separate low-overhead factory method. In the next step, these simple factory methods are merged into increasingly more complex ones until a user-defined overhead threshold is reached. In the last step, a new class is created for each object factory, and constructor calls are rewritten to invocations of the factories' methods. Each step is explained in more detail in the following sections.

As previously illustrated in Chapter 4, the effectiveness of any algorithm that creates object factories greatly depends on the actual types assigned to local variables in a program's intermediate representation. More abstract types generally result in fewer factories that create objects of more different types, and as a result, have a greater potential of confusing pointer analyses. In what follows, we therefore assume that the program has been preprocessed using the type inference algorithm described in Section 4.1.1, which assigns to each local variable the most abstract type possible.

6.4.1 Collecting Information

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

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

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

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

From R it computes the return type r of the factory methods as the most abstract type, i.e., the type with the most subtypes:

$$r = \underset{r_i \in R}{\arg\max}\ |t_s(r_i) \cap j(\mathsf{C})|.$$

Note that the reason why we compute the return type of the factory methods as r, instead of simply choosing X has already been explained in Section 4.2. We therefore do not discuss it in more detail here. Once r has been determined, the set of classes D whose constructors the factory should be able to call is computed as $D = t_s(r) \cap j(\mathsf{C})$.

The tuple (r, D) now uniquely identifies the factory class that will contain the method that should be invoked to replace the object creation o. To add this information to $\mathfrak{F}$, the algorithm operates as follows.

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

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

At the end of the current step each tuple (K, O) represents a factory method that invokes exactly one constructor. In the next step, our new object factory insertion transformation merges pairs of these tuples to create more effective factory methods that invoke multiple constructors.

6.4.2 Merging Factory Methods

The algorithm we developed to merge factory methods is shown in Algorithm 6.2. It makes use of the following definitions.

- Let e be the function for which $e(o)$ gives the number of times the object creation o is executed on average.
- Let $u : \mathbb{T} \mapsto \mathbb{T}$ be the function that maps any reference type to java.lang.Object, the primitive types boolean, byte, char, short, and int to int, and all other types to themselves.

- Let $\mathbb{K}$ be the set of all constructors in the application.

- Let $p : \mathbb{K} \mapsto \mathbb{T}^*$ be the function for which $p(k)$ gives the parameter type list of constructor k, and let $p_u : \mathbb{K} \mapsto \mathbb{T}^*$ be the function for which $p_u(k)$ gives the parameter type list $p(k)$ in which each type t has been replaced with $u(t)$.

- Let $m_u : \mathcal{P}(\mathbb{K}) \mapsto \mathbb{T}^*$ be the function for which $m_u(K)$ gives the parameter type list obtained after merging all parameter type lists in the set $\{p_u(k) \mid k \in K\}$.

- Let a be the number of arguments each factory method uses to determine which constructor to invoke.

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

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

Algorithm 6.2: Factory method merging.

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

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

function c, which is defined as follows.

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

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

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

6.4.3 Generating Factory Classes

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

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

For our running example from Figure 6.1(b) our new algorithm may generate the factory class shown in Figure 6.8, as described above. Note that the only reason that the create method of this class requires three parameters of type java.lang.Object is because the constructor of class X2 requires three arguments of instance types. If that constructor required fewer such arguments, the factory method would also require one argument less. To keep the number of factory method parameters low, it is therefore important that each constructor only requires as many parameters as absolutely necessary. In light of this, the next section discusses how class hierarchy flattening can be improved to significantly reduce cases in which a distinguishing parameter needs to be added to the constructors during flattening.

6.4.4 Improving Class Hierarchy Flattening

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

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

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

As an example, consider the constructor X1(A,B) from Figure 6.1(a). When this constructor is copied to class X2 using the original class hierarchy flattening algorithm, an extra parameter of type D is added to it, because class X2 already declares a constructor with parameter type list [A,B]. Our improved algorithm for CHF, by contrast, will not add a distinguishing parameter if there exists a permutation of the parameter type list [A,B] for which X2 does not have a corresponding constructor. This is the case for permutation [B,A], so the constructor X1(A,B) will be copied as X2(B,A) instead of X2(A,B,D). As a result, OFI will be able to create a factory method that requires one parameter of type java.lang.Object less than the factory method shown in Figure 6.8.

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

6.5 Evaluation

For our evaluation in Chapter 5 we implemented CHF, IM, MM, and OFI as part of an obfuscator we built on top of Soot. To evaluate the techniques

	# application		# transformable	app. size (MB)	
	classes	interfaces	classes (CHF)	pre IO	post IO
avrora	1836	83	1657 (90%)	4.1	2.9
batik	3787	856	3383 (89%)	12.5	9.3
eclipse	5213	1261	3886 (75%)	25.7	17.2
fop	4033	446	3105 (77%)	11.0	8.8
h2	1843	78	1454 (79%)	9.3	7.0
jython	3702	166	941 (25%)	11.8	10.6
luindex	605	28	510 (84%)	1.9	1.2
lusearch	608	28	510 (84%)	1.9	1.2
pmd	1999	451	1507 (75%)	5.6	4.4
sunflow	679	59	557 (82%)	2.0	1.6
tomcat	2173	268	1538 (71%)	10.1	7.1
xalan	2460	426	2111 (86%)	9.6	7.6

Table 6.1: Overview of DaCapo 9.12-bach benchmarks before and after Identifier Obfuscation (IO).

discussed in this chapter, we extended this obfuscator's implementation of CHF with the improvement discussed in Section 6.4.4, and we added implementations of the improved versions of MM and OFI.

Before we continue, it is worth noting that the improvements presented in this chapter do not affect which classes, methods, and fields can be transformed. They also do not fundamentally change how classes and their members are transformed. The transformations presented in this chapter use the same set of basic operations as the transformations discussed in Chapters 2-4, albeit in a different order, or under different circumstances. As a result, the same limitations and restrictions apply, and all previously asserted statements with respect to correctness, maintainability, and reliability remain valid.

6.5.1 Benchmarks

We evaluated our implementation of the techniques we present in this chapter on benchmarks from the *9.12-bach* release of the *DaCapo* benchmark suite [14]. To allow for easier comparison, we use the same benchmarks as in Chapter 5. Table 6.1 gives an overview of them.

Below we present the results obtained using seven different configurations of our obfuscator. First, as a baseline for our comparison, we use the original version of the benchmark from the DaCapo suite, but with field and method identifiers obfuscated. This obfuscation technique is also applied in all other configurations. Doing so is important to make meaningful application size comparisons, as identifier names make up a large portion of the total size of an application. This is confirmed by the two right-most columns of Table 6.1, which show rather large differences in application size before and after identifier obfuscation.

For the second configuration, we generated benchmark versions using CHF, MM, and OFI, but without interface merging. For the remaining five configurations, we used CHF, MM, OFI, and IM with thresholds 10, 20, 30, 40, and 50, respectively. The threshold for interface merging indicates how aggressively it should be applied. It corresponds to the maximum number of classes by which each interface can be implemented before merging stops. For all configurations involving OFI, we set τ to 3.

For each configuration, except the baseline, we generated ten different versions of each benchmark obtained using ten different random seeds. The random seeds are used to group interfaces during IM, and to break ties during MM and OFI. In the next sections, we report average results for those ten versions. As in Chapter 5, the method bodies of dummy methods simply consist of a `return` statement.

6.5.2 Overhead

Figure 6.10 and Figure 6.11 show the overhead of our transformations in terms of application size and execution time, respectively. Results obtained using the techniques described in Chapters 2-4 are shown as diamonds. As shown in Figure 6.10, our improved method merging strategy greatly outperforms the old one. For an IM threshold of 50 the application size overhead is reduced by 31% on average. The largest reduction was observed for eclipse, for which the overhead dropped from 525% to 295%, which amounts to a reduction of 44%. Furthermore, whereas heavily obfuscated versions of half of the benchmarks were originally more than five times larger than the original applications, almost all versions of the benchmarks are now less than four times larger. The only exception is batik, for which the size is 4.1 times larger.

Figure 6.11 shows the median relative execution time of obfuscated versions of our benchmarks during steady-state execution. In this stage of the execution the virtual machine has performed most of its optimizations, and the bytecode is no longer interpreted. As a result, the execution times measured in this phase most closely resemble the actual execution times of the applications, as there is little overhead from the virtual machine. To be able to compare the results of the techniques presented in this chapter to the ones in Chapters 2-4, we measured the ex-

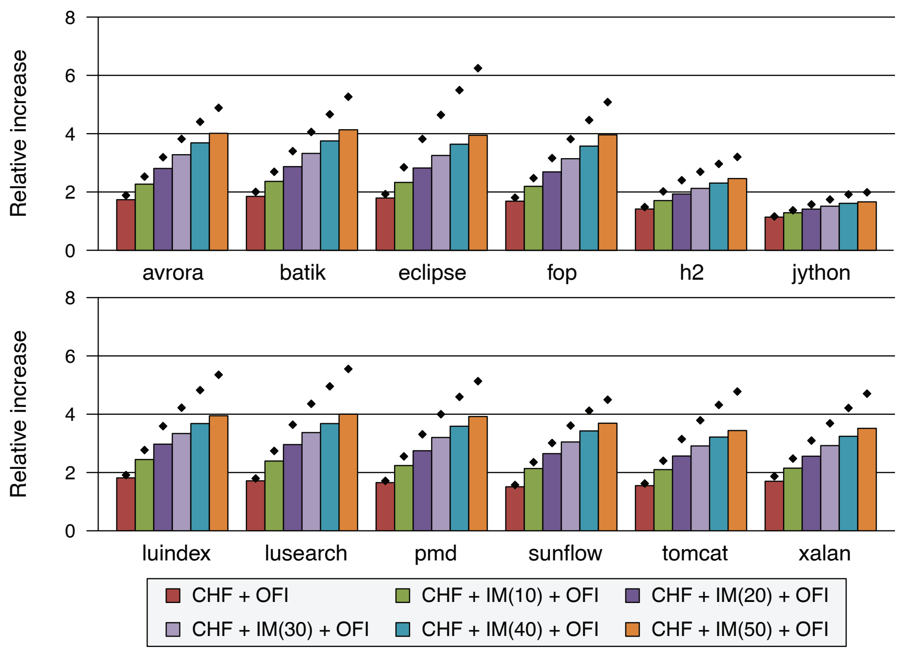


Figure 6.10: Application size overhead of the obfuscations. All values are relative to the original benchmarks.

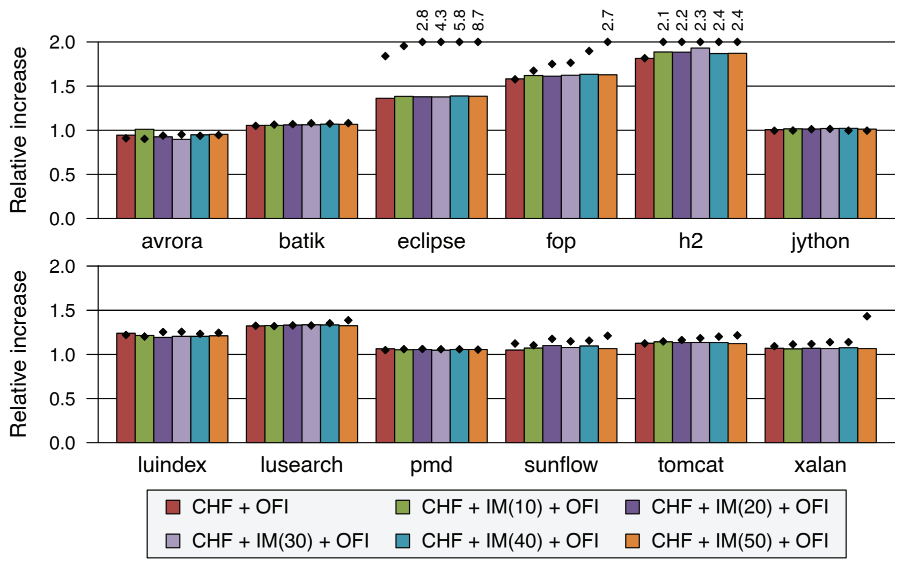


Figure 6.11: Median relative execution time overhead of the obfuscations.

ecution times of all benchmark versions on computing nodes consisting of dual-socket Intel Xeon L5520 processors with 12 GB of memory.

For nearly all versions of eclipse, fop and h2, and some versions of xalan we obtained considerable improvements in execution time. Especially for eclipse, the obtained reductions in execution time are very large. For the most heavily obfuscated versions of this benchmark, our optimizations reduce the execution time overhead by 95%. For lusearch, sunflow and tomcat the absolute improvements in execution time are smaller than for some of the other benchmarks. However, we are still able to reduce their overhead between 16% and 69%, on average.

Most of the performance improvements are a result of OFI, as factory method parameter type lists are on average 65% shorter. However, the improved version of MM also has a positive impact on performance, with parameter type lists of regular methods being 27% shorter, on average.

For some of our benchmarks the improvements presented in this chapter have little effect on their execution time. However, for most of these benchmarks, the execution time overhead of the transformations was already small. Furthermore, as discussed in Chapter 5, some of the overhead is also attributed to CHF, for which we did not present any performance improvements.

In summary, our techniques result in overall size improvements, and more acceptable run-time overheads. Furthermore, because the new object factory insertion technique uses profile information, the overhead of the factories can be controlled better, and is hence more consistent. As a result, we no longer have large variations in execution time as we did for the unimproved versions of our transformations.

6.5.3 Provided Protection

As in Chapter 5, we measured the observable level of protection offered by our techniques using two different metrics. To measure the complexity of transformed applications from the perspective of an automated static analysis tool, we used the average points-to set size as a metric. Large values for this metric imply that the results of other reverse engineering processes, such as virtual call resolution and call graph construction become less precise. Furthermore, additional analysis techniques, such as program slicing [70, 79], that rely on this information also become less precise, and often also slower and more memory-intensive, because more information needs to be tracked and propagated.

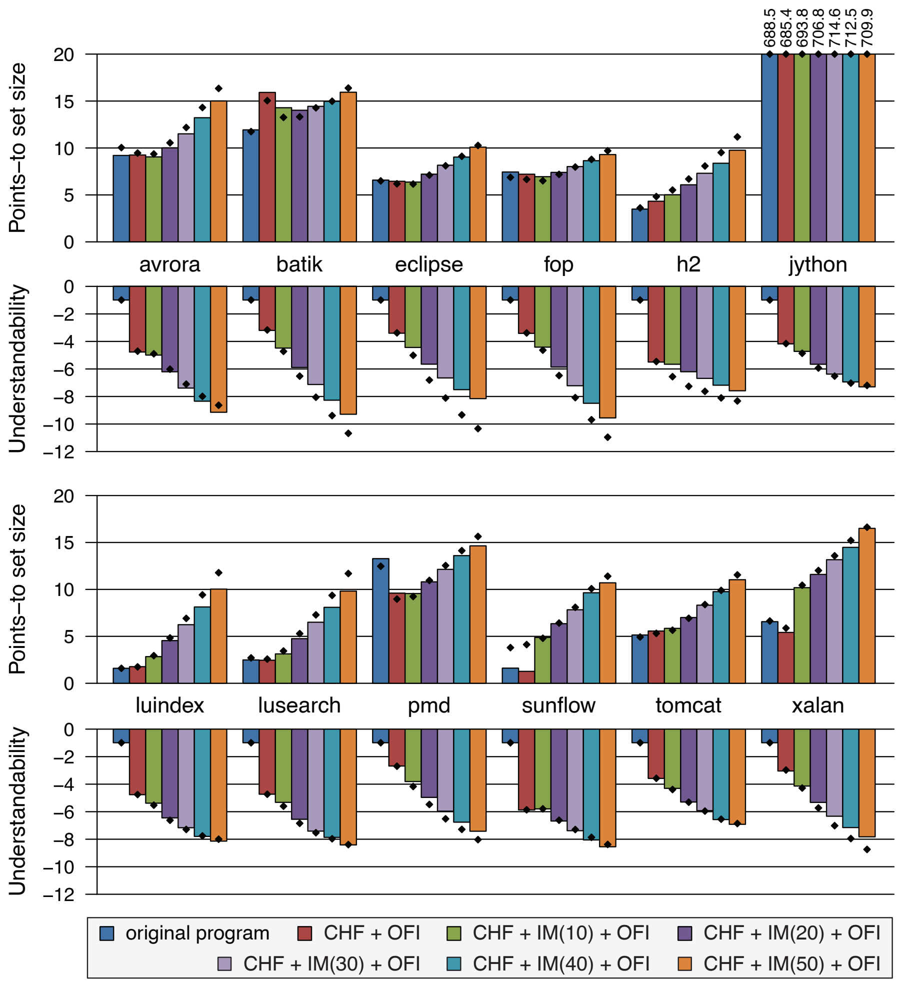


Figure 6.12: Level of protection offered by the obfuscations.

In Figure 6.12, the graphs above the x-axis shows the average points-to set size of the different benchmark versions, obtained using WALA's 01-container-CFA analysis, the most precise pointer analysis offered by the tool [30]. As shown in this figure, the techniques presented in this chapter cause a small reduction compared to previous results. This reduction is mainly caused by the fact that the transformed programs contain more local variables that correspond to method parameters, and that the points-to set sizes of those variables are smaller. The main culprit of the latter is method merging. Because fewer methods are merged, method parameters and their points-to sets are not merged as aggressively. As a result, more parameters with smaller points-to sets remain, which causes the average points-to set size to decrease.

To measure the complexity of transformed programs from the perspective of a human attacker, we used the QMOOD understandability metric [10], as we did in Chapter 5. In Figure 6.12, the charts below the x-axis give an overview of the understandability scores for each of the benchmark versions relative to their respective baselines. For five of the benchmarks, we obtained understandability scores that are comparable to those obtained using the old techniques. For batik, eclipse, fop, h2, pmd, and xalan, we obtained slightly worse scores. Further research revealed that these scores were caused by the improved method merging technique presented in this chapter, which can reduce the coupling component of QMOOD's understandability metric. The coupling metric measures the average number of classes each class refers to by means of its field and method signatures. In order to be cost-effective, the new method merging algorithm often terminates before the original one would. As a result, the parameter type lists of merged methods are shorter, and include parameters of fewer different types.

For avrora, we observed a slight reduction in understandability. This was caused by an increase in the understandability metric's complexity component, which corresponds to the average number of methods in each class. Again, because fewer methods are merged, more dummy methods remain, which causes the perceived complexity to increase. This effect was most noticeable for avrora. We also observed it for other benchmarks, but in those cases it was often masked by a (larger) reduction in coupling. However, the fact that additional empty dummy methods can lead to increases in the complexity of a program highlights an important flaw in QMOOD's complexity metric. While we do believe that the number of methods influences the complexity of an application to some extent, it is a poor complexity metric in itself.

Chapter 7

Prototyping and Practical Issues

Enabling the transformations discussed in the previous chapters on the applications in the DaCapo benchmark suite [14] required a significant engineering effort. We not only had to make changes to Soot [74], WALA [30], and TamiFlex [16], but we also had to develop several tools to collect additional information required during transformation. In this chapter we give an overview of these tools, and discuss some of the technical challenges we had to overcome.

7.1 Experimental Setup

Our experimental setup consists of two parts. The first part focuses on transforming applications using our obfuscator. The second part focuses on evaluating the protection-wise effectiveness of our transformations, and measuring the overhead of transformed applications.

7.1.1 Transformation

As mentioned in Chapters 5 and 6, we implemented our transformations in an obfuscator on top of Soot, a program analysis and transformation framework for Java. To transform applications correctly and with limited overhead, we use Soot in combination with TamiFlex and several tools we developed to collect and construct the input files for our obfuscator. Figure 7.1 gives an overview of these tools and their required inputs.

To transform an application, say the `h2` benchmark from the DaCapo suite, we first use our unpacking tool to gather all the application's classes in a single directory, such that they can be loaded easily by our obfusca-

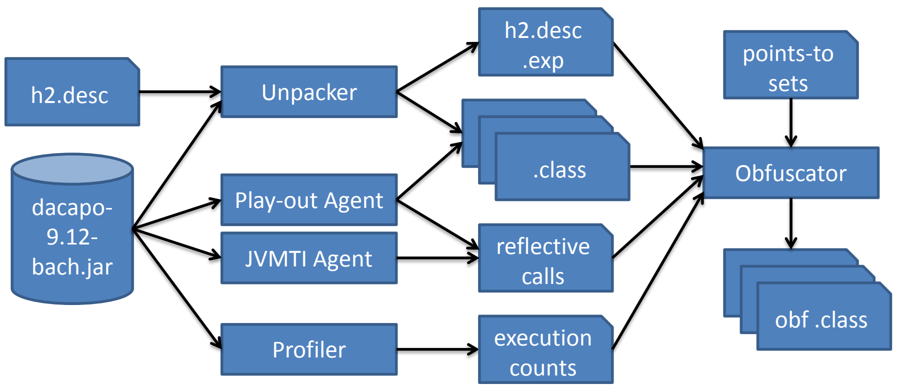


Figure 7.1: Transformation setup

tor. We do this because the DaCapo benchmark suite is distributed as a single jar file that contains a test harness and all fourteen benchmarks. Inside this jar file multiple directories and archives store the inputs for the benchmarks, as well as their code and several libraries they depend on. Note that even though this is an effective way of storing the benchmarks such that they can be invoked and distributed easily, it does make transforming these applications less straightforward.

To gather an application's classes, our tool requires a description file that tells it which directories and archives to unpack. When it is finished, it generates an expanded description file that contains for each directory and archive the classes gathered from it. This file is used by our obfuscator to look up the location of each class in the original program, and determine which classes belong to the same directory or archive.

Next, we use our Java Virtual Machine Tool Interface (JVMTI) agent [58, 60], and TamiFlex's Play-out Agent to collect information about reflective calls performed during the execution of the application. We also instruct the Play-out Agent to output all classes loaded by the virtual machine in the directory in which we unpacked the application. We do this to ensure that any classes that are dynamically generated by the application are also analyzed by our obfuscator.

Third, we use our profiling library to collect the execution counts required by our improved object creation insertion transformation.

Finally, we collect points-to information about the program using WALA. Because WALA is unable to fully analyze programs that contain reflective operations, we first rewrite the application with Booster.

Booster takes two inputs; an application in the form of a set of classes, and a list of reflective operations performed by the application. It outputs a rewritten application (also as a set of classes) in which each reflective operation has been replaced by explicit method calls or operations. We then use WALA to perform a 01-container-CFA points-to analysis on the rewritten program and output the results to file.

When provided with the necessary input files, our obfuscator transforms the application's class files and outputs them to a specified target directory. It also outputs a file (not shown in Figure 7.1) containing information about which class files in the original application should be updated, which new class files were created during obfuscation, and where those classes should be inserted in the application.

7.1.2 Evaluation

We evaluated the effect of our transformations using the setup depicted in Figure 7.2. Given the class files output by our obfuscator we compute several static metrics, such as application size and understandability. To measure dynamic metrics, we first use our packer tool to assemble the obfuscated class files into an actual obfuscated application. Starting from the original application, the set obfuscated class files, the expanded description file, and information from our obfuscator about which new class files were added, our packer tool updates the original application by replacing existing class files by their obfuscated version, and by adding the new class files created by our obfuscator. The result is a new standalone application that can be executed in the same way as the original one. For the resulting application we compute several dynamic metrics, including execution time and memory consumption.

Finally, we use the Play-out Agent to collect a list of reflective calls for the transformed application, such that we can transform it using Booster, and use WALA to compute the points-to sets.

7.1.3 Unpacking and Repacking Applications

The main reason we developed tools to unpack and repack applications, is that Soot is not capable of analyzing and transforming applications, such as those in the DaCapo benchmark suite, that consist of a nested collection of archives and directories. In the past, this problem has been circumvented by using the Play-out Agent to dump all classes loaded

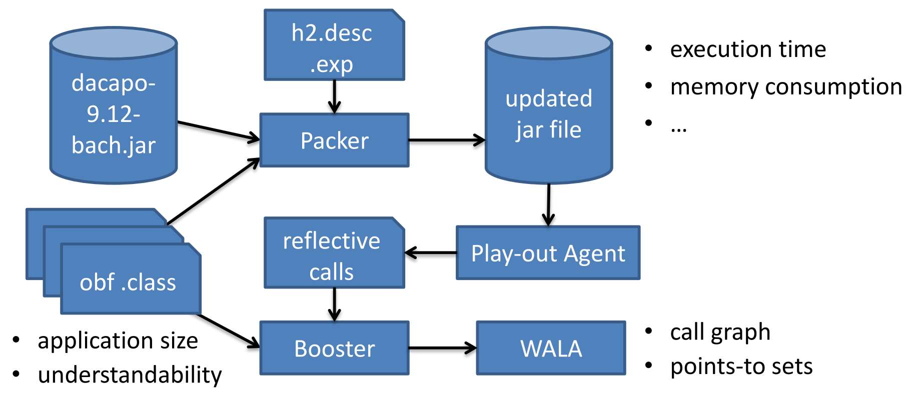


Figure 7.2: Evaluation setup for the h2 benchmark from the DaCapo suite.

during the execution of an application, transform those classes with Soot, and then reinsert the transformed classes at run-time with the Play-in Agent [5, 16]. We opted against this approach for several reasons.

Firstly, our transformations operate under a closed world assumption, which means that all classes must be available at obfuscation time. The Play-out Agent only dumps classes that are loaded during the execution of an application. Hence, relying solely on the Play-out Agent can significantly limit the number of classes Soot has access to, depending on the inputs used when running the application under the agent. The same holds for the list of reflective calls and the set of dynamic classes output by the agent. Whether or not certain calls will be collected, or certain dynamic classes are generated, also depends on the coverage of the inputs. However, in practice it may be much easier to define a set of inputs that cover all reflective calls and all dynamically generated classes (of which there are usually few) than it is to define a set of inputs that causes all classes to be loaded.

Secondly, the Play-out Agent does not provide information about the directories or archives a class was loaded from. This information is important for our transformations to preserve class loader behavior. One may argue, however, that we do not need to take into account the location of each class file during transformation, and that we can simply use the Play-in Agent to load the transformed classes as they are needed. However, even though the Play-in Agent was designed to dynamically replace the definition of a set of classes, using it would not allow us to ignore class loader behavior. The reason is that the agent is implemented as a java.lang.instrument.ClassFileTransformer, which is part of

Java's instrumentation API. Classes that implement ClassFileTransformer are given the opportunity to change the definition of a class after it has been loaded, but before it is actually defined by the virtual machine. Hence, to preserve program behavior the class should already have been loaded by the correct class loader at the point the agent is invoked.

Furthermore, even if the agent would be invoked before the class was loaded, and would hence operate as a class loader, it would still not know when it is allowed to load a specific class from its collection of transformed classes. There may be some unlikely cases where one of the program's class loaders is able to load a class with a certain name, but another class loader is not. Since the task of both these class loaders would be taken over by the Play-in Agent it would be very difficult, if not impossible to automatically determine in which case the agent is allowed to load a specific class.

Thirdly, the Play-out Agent dumps all classes in a single directory, regardless of whether they are application classes or library classes. Without additional information, Soot is unable to determine whether or not a certain class in this directory is a library class that it cannot transform. Even though Soot does allow the user to manually specify by means of inclusion and exclusion rules which classes are library classes and which are not, we found this to be impractical.

Finally, even if we would have all the necessary information to transform the classes and load them correctly using an agent, we would still need to rely on this agent for the program to operate correctly. This makes the application less self-contained, which is not what we want. Note that it is of course possible to include the Play-in Agent as part of the application. However, we chose not to do so. Instead, we developed our packing and unpacking tools to solve all of the above problems.

original use: original classes still there, cannot for obfuscation, maybe empty ones, also by name, not obfuscatable, but we do not do that

Our unpacking tool operates fully automatically, but it does require a description of an application's directories and archives in XML format. As an example, Figure 7.3 shows part of the description file for the h2 benchmark from the DaCapo suite. Note that the file explicitly specifies which jar files to include from the "jar" directory. This is because this directory contains all 50 jar files that make up the different benchmarks. Since we only transform one benchmark at a time, we only select the archives relevant to that benchmark.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE file SYSTEM "ClassContainer.dtd">
<container path="dacapo-9.12-bach.jar">
    <container path="jar/derbyTesting.jar"/>
    <container path="jar/junit-3.8.1.jar"/>
    <container path="jar/h2-1.2.121.jar"/>
    <container path="jar/tpcc.jar"/>
</container>
```

Figure 7.3: User-specified XML-description of the h2 benchmark.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE container SYSTEM "ClassContainer.dtd">
<container path="dacapo-9.12-bach.jar">
    <container path="jar/derbyTesting.jar">
        <classfile filename="FullCollationTests.class"
            name="org.apache.<...>.lang.FullCollationTests"
            package="org/apache/<...>/lang" path=""/>
        ...
    </container>
    ...
</container>
```

Figure 7.4: Automatically generated XML-description of the h2 benchmark.

From the description file in Figure 7.3 our tool knows exactly which directories to traverse and which archives to unpack. As it is doing this, it expands the description file as shown in Figure 7.4. This file contains for each directory and archive which classes they contain, and for each class the name of the file that contains the class file definition, as well as the name of the class, and the package to which it belongs. The path attribute is only required in case the class file does not reside in a directory structure that corresponds to its package. The resulting expanded file serves a dual purpose. Our modified version of Soot uses it to figure out which of the provided classes are actual application classes, and where they are located. The expanded description file is also used by our packing tool to repack the application after transformation, such that it can be executed without the Play-in Agent.

7.1.4 Combining Soot and WALA

To determine the new base types for array creation expressions our class hierarchy flattening transformation relies points-to information about casts and instanceof expressions that involve array instances, which we compute using WALA's 01-container-CFA pointer analysis. Unfortunately, the internal representations used by Soot and WALA are incompatible; our Soot transformations operate on Jimple code, whereas WALA internally represents code in single static assignment (SSA) form [26]. This makes it difficult to link points-to information computed for WALA's SSA instructions to Jimple instructions. Fortunately, when converting a program to its internal representation, both tools tag the instructions in their respective internal representations with the source line number information of the corresponding bytecode instructions. Since there is no need for an obfuscator to preserve the line number information, we remove it, and replace it by specially crafted line number information that we can use to uniquely identify relevant instructions. That way, we can correlate the information computed by WALA on its SSA representation while transforming Jimple code in Soot. After it is no longer needed during transformation, the custom line number information is removed to reduce the size of the application.

7.1.5 Profiling Library

To count how many times each constructor is invoked to create a new object, we instrumented each of our benchmarks statically. We developed a simple tool based around the same set of utility classes that we originally developed for our obfuscator. Our tool prepends each call to an application class constructor used to create a new object (i.e., not a super call) with a call to an instrumentation method in a simple profiling/tracing library. Assuming that class loaders in an application are well-behaved, meaning that they resort to the system class loader when they are not able to load a class themselves, our library can simply be loaded by adding it to the boot class path of the virtual machine.

Note that even though the Java Virtual Machine offers two APIs for dynamic instrumentation, i.e., the JVMTI [58] and the java.lang.instrument API [57], which do not require programs to be rewritten up front, we decided not to use either of them. The reason is that the JVMTI does not provide a means of intercepting object creations in a fine-grained manner. While it is possible to register callbacks for when objects are created and when methods are entered/exited, figuring out whether or not an invocation of a constructor constitutes an object creation or a super call simply from the information passed to the callback routines can be very hard, if not impossible for arbitrary bytecode.

Furthermore, if we had used the java.lang.instrument interface, which enables more fine-grained control over the application, we would have had to create a ClassTransformer to insert the tracing code. In doing so, we would have needed to either re-implement part of our utility classes to work with ASM [17], a bytecode manipulation framework commonly used in combination with ClassTransformers, or find a way to interface with our utility classes, which heavily depend on Soot. We decided to use neither of the approaches, and just instrument the code statically. Even though applications then need to be rewritten before they can be profiled, we feel this is an acceptable compromise, especially given the code that was already available.

Furthermore, because the actual profiling code is not part of the applications, we only had to rewrite the applications once to inject the calls, after which we could develop the profiling library separately.

7.1.6 JVMTI Agent

To complement the output of TamiFlex's Play-in Agent with information about previously unsupported reflective calls, we developed a JVMTI agent to intercept calls to two additional methods: java.lang.Class.getSuperClass() and java.lang.Class.isAssignableFrom(...). Preserving the input-output behavior of these methods is especially important since they query the subtype relations between a program's classes, which may be altered by our transformations if not taken into account. Because these methods have native implementations, we were not able to intercept them in the same way the Play-in Agent does, which is by adding logging calls before the return instructions in the bytecode. Fortunately, it was easy to create wrappers for these methods to log their arguments using the JVMTI.

7.2 Changes to Soot

Soot was an excellent tool to use as a starting point for our obfuscator, because its high-level intermediate Jimple representation enabled us to work at a high abstraction level. This meant that we did not have to take into account any details about the underlying bytecode, unless we absolutely wanted to. Unfortunately, as far as the implementation of our techniques goes, we had little (refactoring) support in Soot to build on. For instance, in Soot it is possible to change the name of a method, and doing so will result in all the necessary internal caches being updated such that the change is reflected in the final bytecode. However, changing the name of a method only does exactly that. The version of Soot from which we started was not capable of computing the set of methods that should keep the same signature as the method whose name is changed, and of updating those methods' names as well.

The authors of JBCO [12], another obfuscator that also features method renaming, had already made an attempt at code to do this. However, we found that their implementation did not always compute the method sets correctly, and that it was difficult to reuse and configure. So rather than trying to fix their implementation, we created our own. Overall, we had to implement most of the refactoring support for our transformations ourselves, including techniques to encapsulate fields, to analyze whether the visibility of class members can be changed, to detect object creations, etc.

Because Soot is under active development, we decided to make as few changes to its original source code as possible. Instead, we built our obfuscator on top of it, and only made changes to Soot if they were unavoidable. Most of these changes are very small and are required to install additional callbacks such that our tool can be informed when Soot has performed certain actions. One of the callbacks installed by our obfuscator notifies it when Soot has finished loading all classes. When this callback is triggered, our obfuscator uses the application's expanded description file to mark each class either as an application class, or as a library class. Unfortunately, it cannot always use the information in this file as is. This is because our transformations assume that all application classes build on a set of self-contained library classes. However, some of the DaCapo benchmarks redefine classes from the standard library. Since these classes are part of a benchmark, they would hence be treated as application classes. For some benchmarks this leads to cases where actual library classes depend on library classes that were redefined as application classes. Since this conflicts with our requirements, we extended Soot to iteratively mark application classes as library classes until the set of library classes is self-contained, essentially turning the application classes into non-transformable classes. Besides reflection, this is also one of the reasons why the number of transformable classes in Table 5.1 is less than the actual number of classes that make up the application.

Other than to add support for callbacks, we also made some changes to Soot to improve the reproducibility of our results. This mainly involved providing implementations of `hashCode` for classes implementing Soot's data structures. Furthermore we added support for several small features needed to transform the DaCapo benchmarks. For instance, we made sure that Soot can handle classes whose names do not match the name of the class file they are defined in.

Finally, we added support for several additional reflective operations whose information was not yet tracked by Soot, and made it easier to track information about new reflective operations. Originally, whenever support for such an operation was needed, code had to be edited in different locations. To avoid this, we refactored the code and removed the parts that were specific to each reflective operation. These parts can now be generated automatically from a simple configuration file that contains a single entry for each reflective operation. Additional reflective operations can now be supported simply by adding a new entry to this file; the supporting code is generated automatically.

7.3 Missing Classes and Interfaces

Many of the DaCapo benchmarks refer to classes and interfaces that are not part of the application itself, nor of the standard library. Special care must be taken when transforming programs that contain such references. Because the referenced types do not exist, they cannot be loaded by any class loader. Furthermore, any classes and interfaces that (indirectly) extend or implement these types can also not be loaded, because the definition of at least one of their supertypes is missing. In what follows, we simply refer to all missing types, and all other classes and interfaces that cannot be loaded because of them, as unloadable types.

To preserve program behavior, our transformations must sometimes treat code that references unloadable types differently than code that does not. This is because certain changes to the code can affect whether or not a virtual machine tries to load an unloadable type, which in turn may affect program behavior. Unfortunately, we cannot avoid this added complexity by removing all references to those types from the application. This is because the behavior of the program may depend on the fact that these types cannot be loaded, or that they do not exist. In rare cases a program may, for instance, expect a ClassNotFoundException or a NoClassDefFoundError when trying to execute a certain code fragment[1].

[1]Note that this would be considered the opposite of good programming practice. However, it is possible in practice, so we need to take it into account.

One case in which we pay special attention to unloadable types is method merging. We can, for instance, not merge two method sets for which the return type of the methods in the first set is void and the return type of the methods in the second set is an unloadable type. If one of the methods in the first set was executed in the original program, the virtual machine will try to load the unloadable type in the transformed program, which may result in a change in program behavior.

Despite taking into account unloadable types during transformation, at some point we were still having problems with the fop benchmark. Transformed versions of this benchmark failed with a ClassNotFoundException for a class that did not exist. We did not expect such behavior because none of the executed code contained any references to unloadable types. While debugging we discovered that the exception was actually triggered by the bytecode verifier, because the transformed program executed without problems when we disabled bytecode verification. By researching the verification process and instrumenting the Java HotSpot Virtual Machine to gain additional insights into the verification process, we discovered that the problem was caused by a combination of our transformations, Soot's Jimple to bytecode conversion process, and the fact that the bytecode of the fop benchmark does not contain StackMapTables.

StackMapTables were introduced in Java 1.6 to speed up bytecode verification by having each method contain type information about its local variables. That way, the bytecode verifier does not have to compute this information using a (slow) iterative data flow algorithm[2]. Instead, a single pass over each method's code suffices. However, because fop's bytecode does not contain StackMapTables, the types of its method's local variables are computed using the iterative data flow algorithm.

This algorithm operates as follows. As long as no fixed point is reached, it propagates the possible types of all local variables and stack locations from instructions to their successors. At instructions that have multiple predecessors, it merges the type information. In doing so, it checks whether there exists a type for all variables and stack locations that is assignment-compatible with all the types computed for the respective variables and stack locations at each of the instructions' predecessors. Depending on the different types of values assigned to each variable, it may be necessary to load additional types to perform these checks.

[2]Even though local variables are untyped in bytecode, the verifier still performs checks to ensure that at all points in the code they hold values of the correct type, such that it can verify that assignments, method calls, etc., are type-safe.

```
static void m(int i){
  A a;
  if(i > 0)
    a = new B();
  else
    a = new C();
  a.m();
}
```

(a) source code

```
   static void m(int i){
 0  iload_0
 1  ifle 15
 4  new B
 7  dup
 8  invokespecial B."<init>":()V
11  astore_1
12  goto 23
15  new C
18  dup
19  invokespecial C."<init>":()V
22  astore_1
23  aload_1
24  invokevirtual A.m:()V
27  return
   }
```

(b) bytecode

Figure 7.5: Source code fragment using a single local variable to hold objects of two different types, and its corresponding bytecode.

It is important to note, however, that additional types are only loaded when potentially conflicting type information needs to be merged. For instance, when all of an instruction's predecessors have computed the same type for a variable, the virtual machine will not load this type, because there is no potential conflict.

To get a better understanding of how the algorithm works, consider the method in Figure 7.5(a). Its corresponding bytecode in Figure 7.5(b) uses a single local variable with index 1 to store the results of new B and new C at bytecode offsets 11 and 22, respectively. At the aload_1 instruction at offset 23 (which has predecessors at offsets 12 and 22) this information is merged, and the type of the variable is computed as A, the lowest common ancestor of types B and C. This information is then further used to verify whether or not the invocation a.m() is type-safe.

Now, consider the method in Figure 7.6(a), which uses two different local variables b and c to store the results of new B() and new C(), respectively. This method can be compiled to a number of different bytecode methods, two of which are shown in Figures 7.6(b) and 7.6(c). The bytecode in Figure 7.6(b) uses a single local variable to hold the instances of classes B and C. This bytecode can be generated because the scopes of b and c do not overlap. However, it is also possible to generate bytecode that uses a separate local variable to hold each of the instances, as shown in Figure 7.6(c). In that case, the result of new B is stored in

```
static void m(int i){
  if(i > 0)
    B b = new B();
  else
    C c = new C();
}
```

(a) source code

```
static void m(int i){
 0  iload_0
 1  ifle 15
 4  new B
 7  dup
 8  invokespecial B."<init>":()V
11  astore_1
12  goto 23
15  new C
18  dup
19  invokespecial C."<init>":()V
22  astore_1
23  return
}
```

(b) bytecode version 1

```
static void m(int i){
 0  iload_0
 1  ifle 15
 4  new B
 7  dup
 8  invokespecial B."<init>":()V
11  astore_1
12  goto 23
15  new C
18  dup
19  invokespecial C."<init>":()V
22  astore_2
23  return
}
```

(c) bytecode version 2

Figure 7.6: Source code fragment using separate local variables to hold objects of two different types, and two of its possible bytecode versions. In (c) the change compared to (b) is shown in red.

local variable 1, whereas the result of new C is stored in local variable 2.

Even though both bytecode versions seem semantically equivalent, they cause different verifier behavior when class A is missing. In that case, the bytecode in Figure 7.6(b) results in an error during verification, whereas the code in Figure 7.6(c) does not. The reason is as follows. With the exception of the call to a.m(), the bytecode in Figure 7.6(b) is identical to the code in Figure 7.5(b), which means that type information about its local variables is propagated and merged in the same way. As a result, the bytecode verifier will try to load class A when merging the type information for local variable 1 at bytecode index 23, just like it would for the code in Figure 7.5(b). However, for the bytecode in Figure 7.6(c) no type information needs to be merged at the return instruction, because variables 1 and 2 can only each contain instances of a single type. As a result, the lowest common ancestor of classes B and C does not need to be computed, and the virtual machine will not try to load class A.

The behavior of an application may hence change depending on whether or not (and where) the local variables and stack locations of its methods are reused. In our case, the behavior of the fop benchmark changed because two local variables in the original bytecode were merged into a single one after transformation, which made the virtual machine try to load a non-existing supertype[3]. The original variables were merged as a result of the fact that our transformations sometimes introduce additional local variables, and because they operate on Jimple code, instead of directly on bytecode. The reason is as follows. Before Soot hands over control to our transformations, it first loads all provided class files and converts their bytecode to Jimple code. In doing so, it converts all local variables and stack locations used in the bytecode to Jimple local variables. Before exiting, Soot converts the Jimple code back to bytecode, in which case it translates the Jimple local variables back to local variable and stack locations in the bytecode. Because our transformations sometimes introduce additional variables, they may affect which Jimple variables are mapped to the same variables in bytecode. As a result, they can unintentionally affect whether or not the verifier will try to load unloadable types, even if the part of the code they modify does not contain references to such types.

However, even if the Jimple code is not touched by our transformations, the generated bytecode may still differ from the original bytecode. This is because Soot performs several optimizations on the Jimple code before converting it to bytecode. One of the optimizations is local variable packing. This optimization tries to reduce the number of local variables needed in bytecode by trying to reuse the same local variables in the bytecode for different Jimple local variables, similar to register allocation in traditional compilers [7]. As is done during register allocation, an interference graph is constructed based on the live ranges of the variables. Whenever two variables do not interfere, they can be mapped to the same bytecode local variable.

To prevent the local variable packer from merging variables of unloadable types with other variables, we modified the algorithm to create artificial interferences between each Jimple local variable declared of an unloadable type and all variables of other types. That way all Jimple variables of unloadable types are mapped to their own local variable in bytecode. As a result, the virtual machine will not try to load unloadable types during bytecode verification. Admittedly, even though this solved the problem in our case, it does not solve the problem in general. For

[3]As a comparison, a similar problem would occur when the bytecode in Figure 7.6(c) is transformed into the code in Figure 7.6(b), assuming that class A is missing.

instance, assume that a program relies on the fact that the method in Figure 7.6(b) cannot be verified when class A does not exist. When this method is transformed into the one in Figure 7.6(c), the behavior of the program changes. Note that even though we expect such cases to be rare in practice, a more general approach to this problem is needed.

One such approach (which we did not implement or evaluate) could be the following. For bytecode that contains references to unloadable types, we would still allow Soot to generate Jimple code, so that all analyses and transformations that operate on such code can still be used. However, in that case we should also instruct Soot to treat such Jimple code as non-transformable, and make sure that it outputs the original bytecode for that Jimple code, instead of its own optimized version.

As a side note, it is worth mentioning that in the case of Figure 7.6(b) there is actually no real need to merge the type information at the return instruction, since the original variables in the source code program would have been out of scope at that point. However, the bytecode verifier still merges the type information because it does not take into account which instructions may follow (if any), nor does it know the source code for which the bytecode was generated. As far as it is concerned, the bytecode could be generated for the method in Figure 7.5(a), in which case the type information needs to be merged.

All of this changes when the code is compiled to contain StackMapTables. In that case, the bytecode actually contains some information about how the variables are used in source code. The corresponding instructions of the methods in Figures 7.5(a) and 7.6(a) can still be identical, with the exception of the aload_1 and invokevirtual instructions required to implement the call a.m(). However, their respective StackMapTables will contain different information. The StackMapTable for the method in Figure 7.5(a) will indicate that the type of local variable 1 at the instruction at offset 23 has type A, like the source code variable. By contrast, the StackMapTable for the method in Figure 7.6(a) will contain no such information, from which the verifier can deduce that the corresponding source code variables went out of scope, and that the type of the variable does not need to be merged because it no longer matters. As a result, neither of the bytecode versions shown in Figures 7.6(b) and 7.6(c) would result in an exception during verification when class A is missing.

It is important to note that, even though StackMapTables can be used to avoid changes in the behavior of the bytecode verifier for our simple example program, we have not investigated whether or not they can be used to avoid this problem in general, nor have we looked into how program transformation frameworks should treat the StackMapTables of methods whose code contains references to unloadable classes.

7.4 Changes to TamiFlex and WALA

During development we made several changes to TamiFlex's Play-in Agent and Booster. We primarily extended the Play-in Agent to add support for additional reflective calls as needed to transform the DaCapo benchmarks in a behavior-preserving way. Our incentive for extending Booster was mainly to obtain more precise call graphs and points-to analyses when using WALA. Despite using WALA in combination with Booster, we were at first not getting the expected results. More specifically, the call graphs computed by WALA did not always include methods of which we were sure they were executed at run time. Normally, one would expect that the static call graphs computed by WALA should at least cover the dynamic call graphs constructed during execution. This was not always the case, however.

To track down the problem more easily, we created a Java Virtual Machine agent based on the JVMTI. We opted for this interface because it can be used to trace method calls by registering callbacks for MethodEntry and MethodExit events. Using our agent we could easily construct dynamic call graphs, which we could then automatically compare to the static call graphs constructed by WALA to detect missing nodes and edges. From the differences between the call graphs we were able to quickly diagnose why some nodes and edges were missing, and fix several bugs in Booster and WALA.

To improve WALA's analyses we extended Booster to add explicit calls to the static initializer methods of classes at all program points where the virtual machine could potentially invoke them. We did so to ensure that static initializer calls are also modeled correctly by WALA. Admittedly, using Booster to perform this task is somewhat discouraged, because applications are not supposed to call the static initializers of their classes directly. However, in practice this is not really a problem, since we never plan on executing the boosted applications, and only use them for static analysis with WALA. Still, in the future it may be worth the effort to investigate how the edges can be added directly in WALA, such that they are also present in the call graphs of programs that have not been rewritten by Booster. In our case it was just easier to add the calls in Booster, because we had to use Booster to inline the reflective calls in our benchmarks anyway, and because we were more familiar with the code base of Booster, which is based on Soot.

7.5 Evaluation and Debugging

Over the years we have generated and evaluated many thousands of transformed program versions. Because generating and evaluating all these applications was a resource-intensive task, we relied on the Flemish tier-1 Supercomputer[4] to carry out some of our experiments. To coordinate our experiments, we built a framework. Based on a configuration file that specifies which applications should be used as input, which transformations should be applied, and which metrics should be computed, it automatically runs all necessary experiments on this supercomputer. Our framework submits experiments in the form of jobs to a cluster of computers. It allows modeling the dependencies between jobs, such that jobs can wait for each other to complete, or run simultaneously. For instance, we can enforce that a program must first be transformed before its execution time is measured. However, the latter can be done in parallel with measuring its memory consumption, or running Booster and WALA on it. For easy post-processing our framework outputs the results of all jobs to an SQLite database.

Even though we were able to solve many problems by analyzing dumps of the Jimple code, we also had to do a fair amount of run-time debugging. For this we used a combination of off-the shelf debuggers and tools we wrote ourselves. The latter were mostly modified versions of the virtual machine agents we had already written to perform certain logging or tracing tasks. One particular debugging method that turned out to be very effective was intercepting all exceptions thrown by an application during its execution and comparing the stack traces of these exceptions before and after transformation. In many cases the exceptions thrown by a program are caught at one point or another during execution, in which case the user is not presented with an exception stack trace. Even if the user is presented with a stack trace, it does not always contain relevant information, because multiple exceptions may have occurred before the program finally crashed. In those cases the actual cause of the problem can often be found by looking at the stack traces of all the exceptions thrown during the execution of the program.

[4]https://www.vscentrum.be/

While performing research presented in this dissertation, we not only modified our agents for debugging purposes. We also adapted them to compute several overhead metrics, such as the total dynamic number of arguments over all methods, as well as the heap and non-heap memory used by an application. Many of the results obtained using these modified agents are presented in Chapter 5 and in Section 6.5.

7.6 Open Sourcing

Many of our bug fixes for Soot, WALA, and TamiFlex have already been made public. The source code of all other modifications and extensions to these tools, as well as the source code of all tools and scripts that we developed to perform the research presented in this dissertation can be found at `http://gujto.elis.ugent.be`.

Chapter 8

Conclusions & Future Work

8.1 Conclusions

In this dissertation we presented class hierarchy flattening, interface merging, and object factory insertion, three obfuscations for object-oriented programs written in managed programming languages. With a prototype tool initially featuring unoptimized versions of these transformations, we obfuscated complex real-world Java applications from the DaCapo benchmark suite that feature reflection and custom class loaders. Our experiments showed that all three transformations are needed in order to achieve good results. Combined, our transformations 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.

Using the unoptimized versions of our transformations, obfuscated program versions were typically, but not always, obtained with low performance and memory footprint overhead, but at a significant application size overhead of up to a factor 6. By allowing users to specify a threshold for interface merging, we provided a simple way of trading off overhead for protection. However, the large variations in performance overhead observed for different versions of some benchmarks with similar levels of obfuscation, and the lack of application size improvements through method merging for one benchmark, suggested that a more effective approach for reducing our obfuscations' overhead was needed, in the form of more controlled and tuned heuristics.

To fulfill these needs, we proposed improved versions of class hierarchy flattening, method merging, and object factory insertion. More specifically, we reduced the number of parameters of constructors copied during flattening, and modified method merging to reduce the variation in the descriptors of merged methods. Furthermore, we developed a model to efficiently and accurately estimate the effect potential method merging operations will have on the size of an application under transformation. Using this model, we are able to stop method merging at the exact point at which it is no longer beneficial.

To reduce the overhead of object factory insertion, we replaced the initial version of the algorithm by an iterative, greedy, profile-driven algorithm that is able to divide classes' constructors over multiple factory methods, instead of combining all of them in a single one. This new algorithm can be tuned by means of a parameter that specifies by how much the dynamic number of arguments required to create objects is allowed to increase. Based on the value of this parameter the algorithm strategically chooses which constructors to place in the same factory methods, not to exceed the user-specified overhead threshold.

We implemented the improved versions of class hierarchy flattening, method merging, and object factory insertion in the same prototype obfuscator that features the original versions of these techniques, and evaluated them on the same set of applications from the DaCapo benchmark suite. Our results showed that our improved techniques offer comparable levels of protection as their original versions, but at much less overhead. On average, our improvements reduce application size overhead by over 30%, and execution time overhead by over 40%. Compared to their original versions, our improved techniques can now be used not only to offer comparable levels of protection at lower overheads, but also to offer more protection at similar levels of overhead.

8.2 Future Work

We envision future work in several directions: stronger protection, reduced overhead, more extensive evaluation, and extended support.

8.2.1 Stronger Protection

First, we see a lot of potential in alternative interface merging 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, interface merging 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.

Secondly, we can improve the effectiveness of method merging as an obfuscation technique. For now, this technique was mainly used to remove dummy methods as a means to reduce application size. As a result, we did not pay much attention to the dummy values provided for the extra parameters generated during method merging. Instead, the values for these parameters are chosen as either zero or null. However, whenever null is used as a parameter to a method, instead of a variable that may actually point to an object, we miss out on an opportunity to increase the points-to set size of that parameter. Hence, the points-to set sizes reported in this work may be lower than the maximum sizes that can be achieved in practice.

Additionally, we can also fill dummy methods that remain in the program after method merging with code copied from different parts of the application, or with randomly generated valid bytecode. This technique can be used not only to confuse static analysis tools that, e.g., compute call graphs or points-to sets, but also to create artificial differences and similarities between classes to improve the resilience of class hierarchy flattening against diffing tools. It is important to note, however, that filling dummy methods in this manner may result in significant increases in application size. Assuming that we fill each dummy method with instructions for an average of 66 bytes per method[1], we expect increases in application size of between 8% for lightly obfuscated applications (no IM) and 50% for heavily obfuscated applications (IM(50)), on average. To reduce this overhead while still filling dummy methods it may hence be more effective to outline existing code into the dummy methods. This will not only require us to fill fewer dummy methods with arbitrary code, but it will also make the dummy methods more useful, as they will actually be invoked during program execution.

Furthermore, we can also extend method merging to allow it to merge pairs of non-dummy methods, similar to the method interleaving transformation presented by Collberg et al. [22]. A combination of each of the method bodies' unused parameters can then be used to decide which of the original method bodies should be executed. Additionally,

[1]The average size of a Java method as reported by Collberg et al. [24].

CHF can be combined with other transformations that operate on the type hierarchy of applications. False factoring [22] can be used to create a fake hierarchy after the existing hierarchy has been flattened, while class coalescing [66] can be used to merge classes after flattening. Using a combination of these techniques and ours, it is likely that applications can be obfuscated even more effectively.

8.2.2 Reduced Overhead

An initial optimization that can be performed after class hierarchy flattening is the removal of abstract classes. Since these classes cannot be instantiated, and other classes do no longer inherit from them, they can often be removed.

To further reduce the size of applications, we can also limit the maximum number of copies that can be created of each method body. A simple way of doing this, is by outlining the code of each instance method that is to be copied during flattening into a new static method that takes a reference to the this object as an explicit parameter. In doing so, the class hierarchy flattening transformation will not copy the methods' original code several times. Instead, it will copy the (often much smaller) code that invokes their original code. Then, after flattening, we can create as many (diversified) copies of each outlined method m as allowed, and replace calls to m by calls to any of its copies.

Even though this technique may effectively reduce code size, it may also degrade performance, because of the extra level of indirection that is added. However, since our transformations result in more application size overhead than execution time overhead, it may be worthwhile to at least partially trade off these overheads by applying this technique to large methods that would otherwise have to be copied many times.

Finally, merging pairs of non-dummy methods as described in the previous section may also help reduce code size. For instance, in some cases it may be beneficial to merge a few pairs of non-dummy methods if also many dummy methods would be removed in the process. Of course if this approach is chosen, our method merging algorithm and our application size model have to be adapted to also compute the cost of the extra decision logic required to select which method body needs to be executed in case two non-dummy methods are merged.

8.2.3 More Extensive Evaluation

For the experiments presented in this work, we only used identifier renaming to obfuscate field and method names, because our tools and transformations were not able to deal with class name and package name changes. The reason is that when we first developed our tools, we did not anticipate that we would ever have to obfuscate class names, so we stored the location of each class as a single path string. In retrospect this was a bad decision, because given these strings, our tools could not always determine correctly which part of the path represents the class' package, and which part represents its actual path. The reason is as follows. Typically, a class a.b.C is stored in a class file C.class, located under a/b/, which results in the path a/b/C.class. However, given a path a/b/C.class, the package name of the class may be a, a.b, or even d.e.f, in case the file C.class actually stores class d.e.f.C.

Because of the large engineering effort required to support class name changes, we have not supported class renaming for a long time. At the time of writing this dissertation, we have refactored our tools, and added support to Soot to track and update the locations of our class files when their name or package changes[2]. Even though class renaming should now be fully supported, we have not had time to sufficiently experiment with this feature to present results as to its impact on application size. That aside, we do expect the application size overhead of our transformations without method merging to be much less, as many descriptors will become shorter. However, despite this, we still expect method merging to be an effective technique for reducing code size.

Furthermore, it may be interesting to perform a sensitivity analysis of the τ parameter that controls the maximum overhead of our improved object factory insertion transformation to see how high it can be chosen before the execution time of transformed programs starts to increase drastically, and what the impact of this parameter is on the ratio of no-arguments constructors to constructors that do require arguments.

Finally, we would like to perform a more extensive evaluation involving more security metrics and actual attack tools. In doing so, it may be interesting to look into automated refactoring techniques that try to improve the class hierarchy of applications [63].

[2]Figure 7.3 already shows our improved description file format.

8.2.4 Extended Support

Our obfuscator currently does not support Java bytecode programs that contain native code. To transform such programs correctly, our transformations need information about which methods, fields, and classes are referenced from the native code, such that they can be treated as non-transformable. Additionally, WALA needs information about which native methods call which Java methods to construct a sound call graph and compute sound points-to sets. At present, we do not have tools that can compute this information, and Soot and WALA are not able to extract this information themselves from the native code.

To collect the required information we can use a dynamic approach, similar to the one used by the TamiFlex Play-out Agent to collect information about reflective operations. We expect this approach to work well, because the native part of a Java application is only allowed to interact with its Java part through a through a well-defined interface known as the Java Native Interface (JNI) [47]. In the same way we created wrapper functions to intercept the arguments and return values of the GetSuperClass and IsAssignableFrom functions as described in Section 7.1.6, we can also create wrapper functions to collect information about all other JNI functions using a custom JVMTI [58] agent.

Alternatively, we could also try to collect the required information statically, using a technique similar to the reflection analysis technique by Livshits et al. [50]. Their technique uses a points-to analysis to determine all the possible sources of strings that are used as class names, in an effort to determine the targets of reflective operations. For instance, by tracking all strings s that can be used as inputs to a call Class.forName(s), their technique is able to determine statically the names of classes that are loaded as a result of the reflective call. Even though their technique works well for reflective Java code, its usefulness may be limited for native code; it can be difficult to track points-to information precisely through the native code, which may forge pointers from memory or involve arbitrary pointer arithmetic.

Another interesting research opportunity could involve applying our techniques to Android applications to complement the transformations currently offered by obfuscators such as Proguard[3] and DexGuard[4]. Overall this should be straightforward, as we can rely on Soot to convert

[3]http://sourceforge.net/projects/proguard/

[4]https://www.guardsquare.com/dexguard

Android dex files to a set of Jimple class files. However, our transformations may need to be updated to take into certain features specific to Android applications, or the Dalvik virtual machine.

Appendix A

Proofs of Lemmas

Lemma 2.1. *$\forall\, b \in \mathbb{B}$. $\overline{e}(b) \subseteq t_s(b')$ after class hierarchy flattening.*

Proof. For rules 2 and 3 this holds, since $\emptyset \subseteq t_s(b)$, and $\{x\} \subseteq t_s(x)$, respectively. For rules 1 and 4 it holds that $\overline{e}(b) \subseteq T$, since $b \in T$, and $\overline{e}(b) \subseteq t_s(b)$ (otherwise b would be non-transformable). After transformation it will therefore either hold that $T \subseteq t_s(i)$, or that $T \subseteq t_s(s)$, and hence that $\overline{e}(b) \subseteq t_s(i)$, or that $\overline{e}(b) \subseteq t_s(s)$, respectively. □

Lemma 2.2. *$\forall\, b \in \mathbb{B}$. $(\overline{h}(b) \setminus \mathbb{T}) \subseteq t^s(b')$ after class hierarchy flattening.*

Proof. For rule 1 it must hold that $(\overline{h}(b) \setminus \mathbb{T}) \subseteq \{$java.lang.Object$\}$. This is always the case, since each class is a subtype of java.lang.Object.

Furthermore, before transformation it holds for each type $b \in \mathbb{B}$ that $\overline{h}(b) \subseteq t^s(b)$ (otherwise b would be non-transformable), and hence also that $(\overline{h}(b) \setminus \mathbb{T}) \subseteq t^s(b)$. Because CHF does not affect the subtype relations between classes outside the subtrees (which are considered non-transformable) and classes inside the subtrees, $(\overline{h}(b) \setminus \mathbb{T}) \subseteq t^s(b)$ also holds after transformation. As a result, the lemma holds for rule 2. For the same reason, the lemma also holds for rule 3, because x belongs to the same subtree as b before transformation.

In case rule 4 applies, a new class s is created for which it holds that $(\overline{h}(b) \setminus T) \subseteq t^s(s)$. Hence it also holds that $(\overline{h}(b) \setminus \mathbb{T}) \subseteq t^s(s)$, since $\overline{h}(b)$ cannot contain any types in $\mathbb{T}$ other than those in T. □

Lemma 2.3. *For each array with base type $b \in \mathbb{B}$ in which an array with base type $c \in \mathbb{B}$ is stored, it holds that $c' \in t_s(b')$ after class hierarchy flattening.*

		c rule 1	c rule 2	c rule 3	c rule 4
	rule 1	(i,i)	(i,c)	(i,x)	(i,s)
b	rule 3	**(c,i)**	(c,c)	(c,c)	**(c,s)**
	rule 4	**(s,i)**	(s,c)	(s,x)	(s,s)

Table A.1: Possible combinations of replacement types for types b and c when an array with base type c is stored in an array with base type b.

Proof. Let a_b be the array with base type b and a_c the array with base type c. To prove this lemma, we will disprove its inverse. In other words, we will prove that our algorithm cannot compute replacement types b' and c' for a_b and a_c such that $c' \notin t_s(b')$. Given that a_c is stored in a_b, Table A.1 shows all possible combinations of types our algorithm can generate for each of the rules that may apply for b and c.

Each cell in the table contains a tuple (b', c') consisting of the replacement types for b and c, given that the rules corresponding to the tuple's row and column apply for b and c, respectively. Note that the table contains no row for rule 2. This is because rule 2 cannot apply for b, since $\overline{e}(b)$ cannot be empty when a_c is stored in a_b. Also note that if rule 3 applies for b, $\overline{e}(b)$ is equal to $\{c\}$, and the replacement type for b is c. Furthermore, if rule 3 applies for both b and c the replacement type for c is also c. This is because when a_c is stored in a_b, $\overline{e}(c)$ is a subset of $\overline{e}(b) = \{c\}$ by construction. Since $\overline{e}(c)$ cannot be empty for rule 3 to apply, it can only be equal to $\{c\}$. The replacement type for is c is hence c itself.

Table A.1 contains three tuples (b', c') for which $c' \notin t_s(b')$ after transformation. These tuples are highlighted in bold. To prove that our algorithm cannot generate such combinations of types, it suffices to prove that if rule 1 does not apply for b, it cannot apply for c (Lemma A.1), and that rule 4 cannot apply for c if rule 3 holds for b (Lemma A.2). Since both these statements hold, the theorem also holds. □

Lemma A.1. *Given an array with base type $b \in \mathbb{B}$ in which an array with base type $c \in \mathbb{B}$ is stored. If rule 1 does not apply for b, it does not apply for c.*

Proof. If rule 1 does apply for b, $\overline{c}(b)$ or $\overline{h}(b)$ contain types that cause conditions 1(a) or 1(b) not to hold. As a result, conditions 1(a) and 1(b) will not hold for c, because $\overline{c}(c)$ and $\overline{h}(c)$ are supersets of $\overline{c}(b)$ and $\overline{c}(b)$, respectively. Rule 1 therefore does not apply for c. □

Lemma A.2. *Given an array with base type $b \in \mathbb{B}$ in which an array with base type $c \in \mathbb{B}$ is stored. If rule 3 applies for b, only rules 2 and 3 can apply for c.*

Proof. Rule 3 can only apply for b if rule 1 does not apply, because rule 1 has higher priority. Consequently, rule 1 also does not apply for c, because of Lemma A.1. Furthermore since rule 3 applies for b, and a_c is stored in a_b, $\overline{e}(b)$ is equal to $\{c\}$. As a result of the way $\overline{e}(b)$ and $\overline{e}(c)$ are computed using Algorithm 2.1, it also holds that $\overline{e}(c) \subseteq \overline{e}(b)$. This means that there are two possible cases for $\overline{e}(c)$.

1. $\overline{e}(c) = \emptyset$, in which case rule 2 applies.
2. $\overline{e}(c) = \{c\}$, in which case rule 3 applies.

Since either rule 2 or rule 3 applies for c, rule 4 does not apply, because it has lower priority. $\square$

List of tables

List of figures

Bibliography

[1] Boomerang - A general, open source, retargetable decompiler of machine code programs. URL `http://boomerang.sourceforge.net`.

[2] Dr. Garbage Bytecode Visualizer. URL `http://www.drgarbage.com/bytecode-visualizer`.

[3] Capstone - The Ultimate Disassembly Framework. URL `http://www.capstone-engine.org`.

[4] Java Decompiler - Yet another fast Java decompiler. URL `http://jd.benow.ca`.

[5] Using Soot and TamiFlex to analyze DaCapo. URL `https://github.com/Sable/soot/wiki/Using-Soot-and-TamiFlex-to-analyze-DaCapo`. [Online; accessed 27-August-2015].

[6] x64dbg - An open-source x64/x32 debugger for Windows. URL `http://x64dbg.com`.

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison Wesley, 1986.

[8] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. ACM. ISBN 0-89791-788-X. doi: http://doi.acm.org/10.1145/236337.236371.

[9] Lee Badger, Larry D'Anna, Doug Kilpatrick, Brian Matt, Andrew Reisse, and Tom Van Vleck. Self-protecting mobile agents obfuscation techniques evaluation report. Technical Report 01-036, NAI Labs, November 2001. URL `http://opensource.nailabs.com/jbet/papers/obfeval.pdf`.

[10] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28 (1):4–17, January 2002.

[11] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *Advances in cryptology-CRYPTO 2001*, pages 1–18. Springer, 2001.

[12] Michael Batchelder and Laurie Hendren. Obfuscating Java: the most pain for the least gain. In *Proc. International Conference on Compiler Construction*, pages 96–110, 2007.

[13] Ben Bellamy, Pavel Avgustinov, Oege de Moor, and Damien Sereni. Efficient local type inference. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 475–492, 2008.

[14] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Communications of the ACM*, 51(8):83–89, August 2008.

[15] Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, pages 3–8. ACM, 2012.

[16] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250. ACM, 2011.

[17] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002.

[18] Jien-Tsai Chan and Wuu Yang. Advanced obfuscation techniques for Java bytecode. *Journal of Systems and Software*, 71(1-2):1–10, 2004. ISSN 0164-1212.

[19] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-box cryptography and an AES implementation. In *Revised Papers International Workshop on Selected Areas in Cryptography*, pages 250–270, 2003.

[20] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.4380250706.

[21] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.

[22] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, University of Auckland, 1997.

[23] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–196, 1998.

[24] Christian Collberg, Ginger Myles, and Michael Stepp. An empirical study of Java bytecode programs. *Software: Practice and Experience*, 37(6):581–641, May 2007. URL `http://dx.doi.org/10.1002/spe.v37:6`.

[25] Oracle Corporation. javap - The Java Class File Disassembler. URL `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html`.

[26] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[27] Markus Dahm. Byte code engineering. In *JIT'99*, pages 267–277. Springer, 1999.

[28] Larry D'Anna, Brian Matt, Andrew Reisse, Tom Van Vleck, Steve Schwab, and Patrick LeBlanc. Self-protecting mobile agents obfuscation report — Final report. Technical Report 03-015, Network Associates Laboratories, June 2003. URL `www.isso.sparta.com/opensource/jbet/papers/obfreport.pdf`.

[29] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. European Conference on Object-Oriented Programming*, pages 77–101, 1995. ISBN 3-540-60160-0.

[30] Julian Dolby, Stephen J Fink, and Manu Sridharan. TJ Watson libraries for analysis (WALA). `http://wala.sf.net`.

[31] Chris Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.

[32] Christophe Foket, Bjorn De Sutter, Bart Coppens, and Koen De Bosschere. A novel obfuscation: class hierarchy flattening. In *Proc. International Symposium on Foundations and Practice of Security*, pages 194–210, 2012.

[33] Christophe Foket, Bjorn De Sutter, and De Bosschere Koen. Pushing Java type obfuscation to the limit. *IEEE Transactions on Dependable and Secure Computing*, 11(6):553–567, November 2014. doi: 10.1109/TDSC.2014.2305990.

[34] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[35] Etienne Gagnon, Laurie Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *Proc. Int. Symp. on Static Analysis*, pages 199–219. Springer, 2000.

[36] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[37] Praneeth Kumar Gone and Mark Stamp. Java design pattern obfuscation. In *Proceedings of the International Conference on Security and Management (SAM)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (World-Comp), 2013.

[38] Li Gong. *Java™SE Platform Security Architecture*.

[39] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23 (6):685–746, November 2001.

[40] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 108–124, 1997.

[41] SA Hex-Rays. Hex-Rays decompiler, 2013.

[42] Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31–55, 2001.

[43] T.W. Hou, H.Y. Chen, and M.H. Tsai. Three control flow obfuscation methods for Java software. *IEE Proceedings-Software*, 153(2):80–86, Apr 2006.

[44] David S Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[45] Min Jae Kim, Jin-Young Lee, Hye-Young Chang, SeongJe Cho, Yongsu Park, Minkyu Park, Philip Wilsey, et al. Design and performance evaluation of binary code packing for protecting embedded software against reverse engineering. In *Object/Component/Service-Oriented Real-Time Distributed*

Computing (ISORC), 2010 13th IEEE International Symposium on, pages 80–86. IEEE, 2010.

[46] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, October 2011.

[47] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999. ISBN 0201325772.

[48] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. ISBN 0201432943.

[49] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, New York, NY, USA, 2003. ACM. ISBN 1-58113-738-9. doi: http://doi.acm.org/10.1145/948109.948149.

[50] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for Java. In *APLAS*, pages 139–160, 2005.

[51] A. Majumdar, A. Monsifrot, and C. Thomborson. On evaluating obfuscatory strength of alias-based transforms using static analysis. In *Proc. International Conference on Advanced Computing and Communications*, pages 605–610, 2006.

[52] Anirban Majumdar and Clark Thomborson. Manufacturing opaque predicates in distributed systems for code obfuscation. In *Proc. Australasian Computer Science Conference*, pages 187–196, 2006.

[53] Anirban Majumdar, Clark Thomborson, and Stephen Drape. A survey of control-flow obfuscations. In *International Conference on Information Systems Security*, pages 353–356, 2006.

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

[55] Peter Molnar, Andreas Krall, and Florian Brandner. Stack allocation of objects in the CACAO virtual machine. In *Proc. International Conference on Principles and Practice of Programming in Java*, pages 153–161, 2009.

[56] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, pages 213–228. Springer, 2002.

[57] Oracle. Java™ java.lang.instrument Package, . URL `https://docs.oracle.com/javase/8/docs/technotes/guides/instrumentation/index.html`.

[58] Oracle. JVM TI Reference, . URL `https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html`.

[59] J. Palsberg, S. Krishnaswamy, Minseok Kwon, D. Ma, Qiuyun Shao, and Y. Zhang. Experience with software watermarking. In *Proc. Annual Computer Security Applications Conference*, pages 308–316, 2000.

[60] CK Prasad, R Ramchandani, G Rao, and K Levesque. Creating a debugging and profiling agent with JVMTI, 2004.

[61] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proc. International Conference on Compiler Construction*, pages 126–137, Warsaw, Poland, 2003.

[62] Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. Java obfuscation approaches to construct tamper-resistant object-oriented programs. *IPSJ Digital Courier*, 1:349–361, 2005.

[63] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1909–1916, New York, NY, USA, 2006. ACM. ISBN 1-59593-186-4. doi: 10.1145/1143997.1144315. URL `http://doi.acm.org/10.1145/1143997.1144315`.

[64] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In *Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 99–110, 1998.

[65] Gregor Snelting and Frank Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, 22(3):540–582, May 2000.

[66] Mikhail Sosonkin, Gleb Naumovich, and Nasir Memon. Obfuscation of design intent in object-oriented applications. In *Proc. ACM Workshop on Digital Rights Management*, pages 142–153, 2003.

[67] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 387–400, 2006.

[68] Mirko Streckenbach and Gregor Snelting. Refactoring class hierarchies with KABA. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 315–330, 2004.

[69] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.

[70] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[71] Frank Tip. Refactoring using type constraints. In *Static Analysis, 14th International Symposium*, pages 1–17, 2007.

[72] Frank Tip and Jens Palsberg. *Scalable propagation-based call graph construction algorithms*, volume 35. ACM, 2000.

[73] Frank Tip, R. Furher, Adam Kieżun, Michael Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *ACM Transactions on Programming Languages and Systems*, 33(3):9:1–9:47, 2011.

[74] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proc. Conference of the Centre for Advanced Studies on Collaborative Research*, pages 125–135, 1999.

[75] Ashok P. Ramasamy Venkatraj. Program obfuscation. Master's thesis, University of Arizona, 2003.

[76] Chenxi Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, University of Virginia, School of Engineering and Applied Science, October 2000. `http://citeseer.ist.psu.edu/wang00security.html`.

[77] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, December 2000. URL `citeseer.ist.psu.edu/wang00software.html`.

[78] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Protection of software-based survivability mechanisms. *dsn*, 00:0193, 2001. doi: http://doi.ieeecomputersociety.org/10.1109/DSN.2001.941405.

[79] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6. URL `http://dl.acm.org/citation.cfm?id=800078.802557`.

[80] Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In *Proc. International Conference on Information Security Applications*, pages 61–75, 2007.