Using Compilation/Decompilation to Enhance Clone Detection

Chaiyong Ragkhitwetsagul, Jens Krinke
University College London, UK

***Abstract*—We study effects of compilation and decompilation to code clone detection in Java. Compilation/decompilation canonicalise syntactic changes made to source code and can be used as source code normalisation. We used NiCad to detect clones before and after decompilation in three open source software systems, JUnit, JFreeChart, and Tomcat. We filtered and compared the clones in the original and decompiled clone set and found that 1,201 clone pairs (78.7%) are common between the two sets while 326 pairs (21.3%) are only in one of the sets. A manual investigation identified 325 out of the 326 pairs as true clones. The 252 original-only clone pairs contain a single false positive while the 74 decompiled-only clone pairs are all true positives. Many clones in the original source code that are detected only after decompilation are type-3 clones that are difficult to detect due to added or deleted statements, keywords, package names; flipped `if-else` statements; or changed loops. We suggest to use decompilation as normalisation to compliment clone detection. By combining clones found before and after decompilation, one can achieve higher recall without losing precision.**

I. Introduction

In this study, we aim to exploit compilation and decompilation as a pre-proessing step for detecting clones in Java programs. A study has shown that compilation/decompilation can enhance performance of 30 code similarity analysers, including clone detection tools [21]. This is because the process of compilation and decompilation canonicalise differences between source code files and can be considered as a code normalisation technique. Similar work is detecting clones after compilation within Jimple code [24], bytecode [4], [14], or assembler code [7]. However, instead of doing clone detection at an intermediate level such as bytecode, Jimple, or assembler level, we use decompilation into Java source code to be able to use **any** Java source code clone detector.

Detecting clones after compilation/decompilation has three major benefits. First, code decompilation generates a second set of source code that can be useful for manual investigation of clones. In our study, we find that some clones discovered after decompilation are interesting and sometimes can be used as a recommendation for code refactoring. This insight cannot be achieved by looking at clones at bytecode or assembler code level. Second, it supports existing state-of-the-art clone detection tools. Since decompiled code is Java source code, one can choose any available Java clone detector. Third, performing clone detection after decompilation can also be used in a case that access to the source code is not available or restricted.

While using compilation/decompilation to augment clone detection has shown promising results, the dataset used in the previous study [21] was limited to 5 small Java programs. They do not represent real environment in software systems with hundreds, or thousands of source code files with third-party APIs and dependencies among classes. This study[1] performs clone detection on three real-world software systems and compares the results before and after decompilation. We resort to the build mechanism provided in each project to handle dependencies in the compilation process, and use a decompiler to retrieve a decompiled versions from the class files. The findings show that using compilation/decompilation to enhance clone detection can be applied to real-world software systems. Furthermore, there are clones that are challenging to detect for clone detectors in the original code but can be discovered after decompilation (see Figure 3 and Figure 4 for examples). This opens a possibility of using decompilation to increase accuracy of clone detectors.

This paper makes the following primary contributions:

1. A study of effects of compilation/decompilation to clone detection: We demonstrate that using compilation/decompilation as a pre-processing step of clone detection is feasible for real-world Java projects. By combining clones found before and after decompilation, one can achieve higher recall without losing precision.

2. Providing insights to decompiled clones: Our manual investigation shows that there are clones which can only be discovered using compilation and decompilation. We summarise their characteristics.

3. Clone oracle: 326 manually validated clone pairs can be used as a clone oracle in future clone studies.

II. Experimental Design

The study aimed to answer the following research questions:

RQ1 (Clone agreement): How many clone pairs are mutually agreed and reported by the same clone detector before and after decompilation? How many clone pairs are exclusively reported before and after decompilation?

RQ2 (Decompilation accuracy): How does compilation/decompilation affect precision and recall of clone detectors?

RQ3 (Characteristics of disjoint clones): What are the characteristics of clones discovered only in the original source code before decompilation? Similarly, what are the characteristics of clones that can be detected only after decompilation?

[1]The results and manually validated clone pairs can be found at http://cragkhit.github.io/crjk-iwsc17.

Accepted for publication by IEEE. © 2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

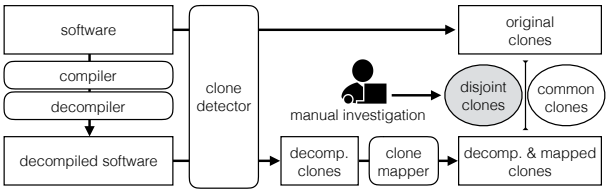

Fig. 1. The experimental framework

TABLE I
SOFTWARE SYSTEMS

System	Version	Original		Decompiled	
		Files	SLOC	Files	SLOC
JUnit	4.13	203	9,777	311	11,233
JFreeChart	1.5.0	644	96,711	669	85,251
Tomcat	9.0	1,688	241,924	2603	256,974

A. Experimental Framework

The framework of the study is depicted in Figure 1. Given a software system, we remove comments and apply pretty-printing to the source code. The system is then compiled and decompiled to generate another version of the software. A clone detector is applied to both versions. This process generates two clone reports: one for the original code, and another one for the decompiled code. We are interested in method-level clones in this study so the clone report contains file names, starting lines and ending lines of cloned method pairs. Since starting and ending line of the clones in the decompiled clone report are different from the original report, we cannot compare decompiled clones to original clones directly. Thus, we build a mapping tool to map starting and ending lines of decompiled clones to their respective locations in the original code and generate another version of the report, *decompiled-and-mapped* clone report. We compare the original and decompiled-and-mapped clone report to find common and disjoint clone pairs. Finally, we manually look at the disjoint pairs to check if they are true clones.

B. Software Systems

We select the latest versions (obtained on 19 November 2016) of three well-known Java open source systems for this study: *JUnit v.4.13*, *JFreeChart v.1.5.0*, and *Apache Tomcat v.9.0* from GitHub. The size[2] of three systems are varied as listed in Table I. Tomcat is the largest project in the set having approximately 240K SLOC. It is 2.5 times bigger than JFreeChart and 25 times bigger than JUnit. We are only interested in Java source code but not test code so we remove all testing class files before the analysis.

C. Tools

1) Compiler and Decompiler: We use the standard *javac* as a compiler and an open-source tool *procyon* [19] as a decompiler. Procyon has advantages over other Java decompilers for its ability to handle declaration of `enum`, `String`, `switch` statements, anonymous and named local classes, annotations, and method references.

[2]The size is measured in terms of SLOC (excluding comments and blank lines) by cloc tool (https://github.com/AlDanial/cloc)

TABLE II
NICAD'S CONFIGURATIONS

Config.	Parameters
Type-1	UPI=0.0, renaming=none
Type-2	UPI=0.0, renaming=consistent
Type-3	UPI=0.3, renaming=consistent

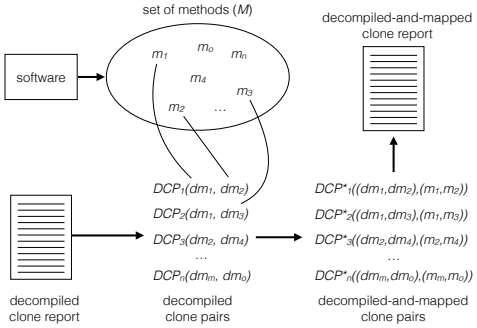

Fig. 2. The process of mapping decompiled clones to their original locations

2) Clone Detector and its Configurations: We select the well-known *NiCad* tool as a clone detector for this study. NiCad has been used extensively in several clone research studies [21], [23], [25], [27]. It can detect clones at method level which suitably supports our clones mapping algorithm. Additional benefit of using NiCad is its ability to detect and categorise clones into type-1, type-2, and type-3 by choosing from its pre-defined configuration files. We select three sets of parameter configurations for NiCad as listed in Table II. The default configuration (UPI=0.3, renaming=none) does not conform to any clone type and is also subsumed by the type-3 configuration so we do not include it in this study. Our method allows other method-level clone detectors such as DECKARD, or SourcererCC to be used if required. However, in this study, we focus more on the effects of decompilation to different clone types rather than comparing different tools and detection approaches.

3) Clone Mapping Tool: In a normal clone detection activity, one runs a clone detection tool against a software system or multiple software systems and consults a clone report to locate clones in the software. In this study we have not only an original software system but also another decompiled version of the software. We implemented a clone mapping tool that automatically processes decompiled clone pairs and maps them back to their original locations. The tool offers several benefits. With the clone mapper, we can compare clones before and after decompilation just by using line numbers. Moreover, after mapping, one can directly incorporate decompiled clones into their original results since their locations are consistent with the original code. Finally, the generated clone report conforms to the format of NiCad clone report and can be analysed by other clone evaluation frameworks based on clone lines (e.g. Bellon's [2], BigCloneBench [26], EvaClone [27]) in the same way as the original.

The overview of the clone mapping process is shown in Figure 2. The tool works at method level. The clone mapping algorithm relies on fully-qualified class name, method name, and its parameters as matching criteria. The clone mapper tool starts by extracting a set M of all methods and constructors from a software system under analysis. A method x is stored as a vector m_x containing a method name, vector p of parameters, starting line, ending line, and fully-qualified class name: $m_x = [name, p, start, end, FQClassName]$. Then, the tool reads a decompiled clone report and extract all decompiled clone pairs (DCP). Each DCP contains two decompiled methods $DCP(dm_x, dm_y)$ reported as clones to each other. Clone mapper iterates over all decompiled clone pairs and tries to match each decompiled method to every original method in M based on *name*, p, and *FQClassName* by string matching. For example, as illustrated in Figure 2, a decompiled clone pair $DCP_1(dm_1, dm_2)$, finds matches between dm_1 and m_1 and between dm_2 and m_2. Then, the clone mapper creates a decompiled-and-mapped clone pair $DCP*_1((dm_1, dm_2), (m_1, m_2))$ containing the clone pair with locations in both decompiled and original source code. If there is no match, that means the matching method does not exist in the original source code and solely generated by the decompiler (e.g. default constructors). The tool ignores such unmatched methods and all its respective clone pairs. After all decompiled clone pairs are processed, the clone mapper generates a *decompiled-and-mapped* clone report from the set of $DCP*$. The decompiled-and-mapped clone report is used along with the original clone report to find common and disjoint clone pairs.

D. Common and Disjoint Clone Pairs

Using the original and decompiled-and-mapped clone report, we extract two sets of clone pairs: C_{orig} and C_{decomp}. We find clone pairs that are common between them by performing a set intersection. We call clone pairs in the intersection *common clone pairs* (C_{common}).

$$C_{common} = C_{orig} \cap C_{decomp}$$

Clone pairs that can only be found in the original ($C_{orig\text{-}only}$) and decompiled set ($C_{decomp\text{-}only}$) are results of subtraction by the common clone pairs. We call them *disjoint clone pairs*.

$$C_{orig\text{-}only} = C_{orig} - C_{common}$$
$$C_{decomp\text{-}only} = C_{decomp} - C_{common}$$

We mainly focus on disjoint clone pairs for manual investigation. This approach gives us clones that are detected only before and after decompilation. By focusing on the disjoint clone pairs, we can reduce the number of clones that need to be manually investigated dramatically and are able to study them in more details.

E. Clone Filtering

Before looking at the clones, we filter the clone pairs using regular expressions to capture and remove clone pairs that are `equals()`, `hashCode()`, getters, setters, and duplicated methods generated by the decompiler. We prune them because they are not very interesting to look at. The `equals()`, `hashCode()`, getter and setter clone pairs are similar boiler-plate code. The duplicated methods are inner-class methods which are by-products from compilation/decompilation process. They must be removed since they do not exist in the original code.

III. Results and Discussion

We perform an experiment on an iMac running macOS 10.12.1 with 2.7 GHz Intel Core i5 and 8 GB of RAM. The answers to the three research questions are discussed below.

RQ1: Clone Agreement

We answer RQ1 by running NiCad against the three software systems twice, before and after decompilation, and studying the clones. NiCad is configured using three different configurations: type-1, type-2 with consistent renaming, and type-3 with consistent renaming (i.e. using NiCad's configuration file *type1.cfg*, *type2c.cfg*, and *type3-2c.cfg* accordingly). NiCad provides blind and consistent renaming options. We choose the stricter consistent renaming so that we can reduce the number of false positives. Then, we use the clone mapper to map decompiled clones back to their original counterparts. Finally, we compute an intersection of clone pairs between the *original* (C_{orig}) and *decompiled* (C_{decomp}) set to find common and disjoint clone pairs. The same approach of finding common and disjoint clones before manual analysis has been also done by Kononenko et al. [14].

The number of clone pairs in common, orig-only, and decomp-only set before and after filtering are displayed in Table III. The set of clone pairs after filtering is denoted as *Cf*. The clones are divided by clone types from type-1 to type-3. The numbers are mutually exclusive. For example, type-2 clone pairs are pairs that are found using type-2 configurations and not reported in type-1 pairs. Similarly, the number of type-3 clone pairs are the ones not reported in type-1 and type-2.

The findings from the three systems are discussed below.

1) JUnit: The system contains no type-1 clone. After filtering, we find 6 type-2 and 3 type-3 clone pairs and all of them are identically reported from both before and after decompilation. We do not find any disjoint clone pairs. So, we do not continue the manual investigation for JUnit.

2) JFreeChart: The followings are numbers after filtering. For type-1 clones, we find 33 (89.2%) common, 1 (2.7%) orig-only, and 3 (8.1%) decomp-only pairs. For type-2, there are 159 (83.2%) common, 15 (7.9%) orig-only, and 17 (8.9%) decomp-only clone pairs. For type-3, there are 155 (67.4%) common, 48 (20.9%) orig-only, and 27 (11.7%) decomp-only pairs.

3) Tomcat: After filtering, we find 20 (46.5%) common, 22 (51.2%) orig-only, and 1 (2.3%) decomp-only clone pairs in type-1. For type-2, there are 217 (88.6%) common, 25 (10.2%) orig-only, 3 (1.2%) decomp-only clone pairs. Lastly, for type-3, there are 608 (78.8%) common, 141 (18.3%) orig-only, 23 (2.9%) decomp-only pairs.

For RQ1, we find that, after filtering irrelevant clone pairs, the clone pairs before and after decompilation are mostly similar for all three clone types. In JUnit, 100% of clone pairs are identically reported before and after decompilation. In JFreeChart and Tomcat, common clone pairs account for 67.4% to 89.2%, and 45.5% to 88.6% respectively. Nevertheless, we still find a significant number of disjoint clones for all three clone types which means there are clones that can avoid the detection before and after decompilation. The number of decomp-only clone pairs in JFreeChart and Tomcat keep increasing from type-1 to type-3. This demonstrates that compilation/decompilation is useful in discovering clones with changes (type-2 and type-3). However, it can only marginally improve the detection of type-1 clones since they are already handled by NiCad pretty-printing.

RQ2: Decompilation Accuracy

We manually investigate 326 clone pairs (252 from $Cf_{orig\text{-}only}$ and 74 from $Cf_{decomp\text{-}only}$) in JFreeChart and Tomcat. The first author takes a role of an investigator. The investigator looked at the clones in the two sets and classified them as either true or false positive. For each clone pair, he checked them both in the original and decompiled version. However, the classification is only based on the original code. He also note the details of the clones before and after decompilation and the reason of why they are reported in only a single set. The manual investigation results are shown in Table IV. We can see that every clone pair, both in the original and decompiled set, is classified as true positive except for a single one in JFreeChart orig-only type-3 clones.

Considering the number of clones and true positive pairs in both $Cf_{orig\text{-}only}$ and $Cf_{decomp\text{-}only}$ set, we can see that NiCad offers perfect precision almost in every setting. However, in terms of recall, NiCad misses a considerable amount of clone pairs that are reported only in the original or decompiled version.

JFreeChart: There are 47 true clone pairs from $Cf_{decomp\text{-}only}$ that are not found in the original version. On the contrary, there are 63 true clone pairs from $Cf_{orig\text{-}only}$ that are not reported in the decompiled version.

Tomcat: There are 27 true clone pairs from $Cf_{decomp\text{-}only}$ that are discovered after decompilation. On the other hand, 188 true clone pairs in $Cf_{orig\text{-}only}$ are missing after decompilation.

To answer RQ2, we find that original and decompiled source code do not have perfect clone recall. However, one can compliment the original clone results by incorporating clones after decompilation. From the manual investigation, we find that all decompiled clone pairs are true positives. Combining two clone sets will increase recall of the tool without losing precision.

RQ3: Characteristics of Disjoint Clones

The manual investigation reveals 7 characteristics of disjoint clones from JFreeChart and Tomcat. The details of disjoint clone characteristics are described in Table V. Three characteristics are found from clones in $Cf_{orig\text{-}only}$ and four are found from clones in $Cf_{decomp\text{-}only}$.

$Cf_{orig\text{-}only}$*:* The majority of the clone pairs here do not have their counterparts after decompilation due to effects of the decompilation process. The most common characteristic is smaller clone size after decompilation. 169 pairs of the original clones are smaller after decompilation. They are smaller than the 10-line minimum clone size of NiCad and hence not reported (making them appear only in the original set). The second characteristic is that clones become more different after decompilation (38 pairs). For example, two methods in the original source code containing a string constant with the same variable name but different values. The variables are declared outside of the clone region thus they form an identical type-1 clone pair. After decompilation, the constant variables have been replaced by the actual value of string literals. This makes decompiled code no longer an identical clone pair. Another characteristic, observed from 5 clone pairs, is a decrease of similarity due to smaller clone size after decompilation. In some cases, a type-3 clone pair with added lines gets smaller after decompilation. The added lines are preserved while other statements are compressed or removed. Thus, the decompiled clone pair has a lower similarity value. The remaining 41 disjoint pairs do not have any noticeable characteristics (categorised as *Unknown*).

$Cf_{decomp\text{-}only}$*:* most of the clone pairs are challenging type-2 and type-3 clones for NiCad. There are 19 clone pairs that in the original code contain added/deleted statements, extra type castings (e.g. `(CategoryAxis)this.domainAxes.get(index)` vs. `this.rangeAxes.get(index)`), or package names in front of class names (e.g. `Map.Entry` vs. `Entry`). The added/deleted statements lower clone similarity while extra type casts and package names affect type-1 and type-2 detection. These inconsistencies are standardised and the clone pairs are more similar after decompilation. Moreover, we observe 15 clone pairs having different `if-else` statements similar to the example depicted in Figure 3. The method *findDomainBounds()* and *findRangeBounds()* form a type-3 clone pair with flipped but equivalent `if-else` conditions. These `if-else` statements are canonicalised by the decompilation process and become identical. Interestingly, this type-3 clone pair can be discovered using even stricter type-2 configurations after decompilation. There are 4 type-3 clone pairs with different loops, `for` and `while`. An example is shown in Figure 4. They turn almost identical after decompilation by having only `for` loops. Lastly, we found 2 clone pairs residing in inner classes. They are missing from the original clone set possibly due to complications in parsing. Compilation/decompilation extracts inner classes out as separated files so they can be easily detected. There are 35 pairs only found after decompilation but without any observable characteristic (categorised as *Unknown*).

For RQ3, we derive 7 characteristics of disjoint clones that make them discoverable only before and after decompilation. We observe that majority of clones reported only in the original set is because of smaller size after decompilation. The decompiled clones are still clones but they are two small too be reported. On the contrary, the characteristics of clone pairs only found by decompilation

TABLE III

SYSTEMS AND CLONES FOUND CATEGORISED BY CLONE TYPES. THE NUMBERS ARE CLONE PAIRS FOUND ONLY IN A PARTICULAR CLONE TYPE (NON-SUBSUMING). C DENOTES "CLONE PAIRS" AND Cf DENOTES "FILTERED CLONE PAIRS".

System	Clone type	$\|C_{common}\|$	$\|C_{orig\text{-}only}\|$	$\|C_{decomp\text{-}only}\|$	$\|Cf_{common}\|$	%	$\|Cf_{orig\text{-}only}\|$	%	$\|Cf_{decomp\text{-}only}\|$	%
JUnit	Type-1	0	0	11	0	0.0	0	0.0	0	0.0
	Type-2	6	0	0	6	100.0	0	0.0	0	0.0
	Type-3	4	0	0	3	100.0	0	0.0	0	0.0
JFreeChart	Type-1	43	1	10	33	89.2	1	2.7	3	8.1
	Type-2	535	42	40	159	83.2	15	7.9	17	8.9
	Type-3	25604	12006	1885	155	67.4	48	20.9	27	11.7
Tomcat	Type-1	24	27	254	20	46.5	22	51.2	1	2.3
	Type-2	270	34	7	217	88.6	25	10.2	3	1.2
	Type-3	790	161	121	608	78.8	141	18.3	23	2.9
Total		27276	12271	2328	1201	78.7	252	16.5	74	4.8

TABLE IV

MANUAL INVESTIGATION RESULTS OF CLONE PAIR CANDIDATES REPORTED IN $Cf_{orig\text{-}only}$ AND $Cf_{decomp\text{-}only}$

System	Type	$Cf_{orig\text{-}only}$		$Cf_{decomp\text{-}only}$	
		Cand.	TP	Cand.	TP
JFreeChart	Type-1	1	1	3	3
	Type-2	15	15	17	17
	Type-3	48	47	27	27
	Sum	64	63	47	47
Tomcat	Type-1	22	22	1	1
	Type-2	25	25	3	3
	Type-3	141	141	23	23
	Sum	188	188	27	27

involve type-2 and type-3 clones with strong modifications at syntactic level. After compilation/decompilation, the modifications are canonicalised.

IV. THREATS TO VALIDITY

The three chosen software systems for our experiment might not be representatives of all Java software projects and the results might not be generalised. We are aware of the effects of configurations to the tools' performance, so we tuned NiCad using multiple pre-defined configurations. At the same time, they are configurations that conform to the definitions of type-1, type-2, and type-3 clones. Nevertheless, we only selected subsets of all possible NiCad configurations. Lastly, there is only a single clone detection tool and decompiler chosen. They might not represent other clone detectors and decompilers.

V. RELATED WORK

Clone detection is an active research topic in software engineering for several decades. Locating duplicated pieces of code provide several insights into software systems. It has applications such as software plagiarism detection [18], source code provenance [6], and software licensing conflicts [9].

Several clone detection tools for source code and binary code have been introduced by the research community. Many of them are based on string comparison techniques such as Longest Common Subsequence (LCS) found in NiCad [22]. Many tools transform source code into an intermediate representation such as tokens and apply similarity measurement on them (e.g. SourcererCC [23], CCFinder [12], CP-Miner [16], iClones [10]). Structural similarity of clones can be discovered by comparing AST as found in CloneDR [1] and Deckard [11] or by using program dependence graphs [13], [15].

Code normalisation enhance similarity measurement of two code fragments by modifying their layouts, identifiers, statements into a standard format or by changing the code into other representations. For example, a normalisation is done by transforming code into an intermediate representation like token streams [12] or abstract syntax trees [1], [11]. NiCad [22] uses TXL with pretty printing as part of the normalisation process. Ragkhitwetsagul et al. [21] shows that compilation/decompilation can also be considered as a kind of code normalisation.

Decompilation converts a program from low-level to high-level language. It is normally used to recovered source code from compiled software artefacts such as bytecode or binary code. There are several studies on decompilation techniques for various languages [3], [5], [8], [17], [20]. Decompilers are useful when source code of a software system is absent or inaccessible. On the other hand, it can be maliciously used to create a cloned or plagiarised program by decompiling an original app, making alterations, and repackaging it. Chen et al. [4] find 13.51% of applications from five different Android markets are clones created by decompiling original apps and repackaging them into new apps.

There have been a few studies similar to ours by trying to detect clones after compilation. Chen et al. [4] located clones in Android apps based on dex files extracted from Android APKs. Davis and Godfrey [7] convert Java and C/C++ code into assembler code and detect clones using longest common subsequence string matching augmented by hillclimbing search for flexible matching. Kononenko et al. [14] similarly find clones in Java after compilation by adapting CCFinderX to be compatible with bytecode sequences and manually investigate disjoint clone pairs. Selim et al. [24] enhance Simian and CCFinderX by transforming Java code into Jimple code and located clones at that level. Their technique helps the tools to detect more type-3 clones and handle gapped clones. Our study detect clones at source code level using the current state-of-the-art code clone detection tool after applying a two-step process of compilation and decompilation. This approach provides opportunities to compare and study clones before and after

```java
/* original code */
@Override
public Range findDomainBounds(XYDataset dataset) {
  if (dataset==null) {
    return null;
  }
  Range r=DatasetUtilities.findDomainBounds(dataset,false);
  if (r==null) {
    return null;
  }
  return new Range (r.getLowerBound()+this.xOffset,
      r.getUpperBound()+this.blockWidth+this.xOffset);
}
```

```java
/* decompiled code */
@Override
public Range findDomainBounds(final XYDataset dataset)
{
  if (dataset==null ) {
    return null;
  }
  final Range r=DatasetUtilities.findDomainBounds
                                  (dataset,false);
  if (r==null) {
    return null;
  }
  return new Range(r.getLowerBound()+this.xOffset,
    r.getUpperBound()+this.blockWidth+this.xOffset);
}
```

```java
/* original code */
@Override
public Range findRangeBounds(XYDataset dataset) {
  if (dataset!=null) {
    Range r=DatasetUtilities.findRangeBounds(dataset,false);
    if (r==null) {
      return null;
    } else {
      return new Range(r.getLowerBound()+this.yOffset,
         r.getUpperBound()+this.blockHeight+this.yOffset);
    }
  } else {
      return null;
  }
}
```

```java
/* decompiled code */
@Override
public Range findRangeBounds(final XYDataset dataset)
{
  if (dataset==null) {
    return null;
  }
  final Range r=DatasetUtilities.findRangeBounds
                                    (dataset,false);
  if (r==null) {
    return null;
  }
  return new Range(r.getLowerBound()+this.yOffset,
    r.getUpperBound()+this.blockHeight+this.yOffset);
}
```

Fig. 3. Example of type-3 clones in *findDomainBounds()* and *findRangeBounds()* that can be detected with type-2 configuration after decompilation

```java
/* original code */
public void clearRangeMarkers() {
  if (this.backgroundRangeMarkers!=null) {
    Set<Integer> keys=this.backgroundRangeMarkers.keySet();
    for (Integer key:keys) {
      clearRangeMarkers (key);
    }
    this.backgroundRangeMarkers.clear();
  }
  if (this.foregroundRangeMarkers!=null) {
    Set<Integer> keys=this.foregroundRangeMarkers.keySet();
    for (Integer key:keys) {
      clearRangeMarkers(key);
    }
    this.foregroundRangeMarkers.clear();
  }
  fireChangeEvent();
  }
```

```java
/* decompiled code */
public void clearDomainMarkers() {
  if (this.backgroundDomainMarkers!=null) {
    final Set<Integer> keys=this.backgroundDomainMarkers
                                              .keySet();
    for (final Integer key:keys) {
      this.clearDomainMarkers(key);
    }
    this.backgroundDomainMarkers.clear();
  }
  if (this.foregroundDomainMarkers!=null) {
    final Set<Integer> keys=this.foregroundDomainMarkers
                                              .keySet();
    for (final Integer key:keys) {
      this.clearDomainMarkers(key);
    }
    this.foregroundDomainMarkers.clear();
  }
  this.fireChangeEvent();
}
```

```java
/* original code */
 public void clearRangeMarkers() {
   if (this.backgroundRangeMarkers!=null) {
     Set keys=this.backgroundRangeMarkers.keySet();
     Iterator iterator = keys.iterator();
     while (iterator.hasNext()) {
       Integer key=(Integer) iterator.next();
       clearRangeMarkers (key.intValue());
     }
     this.backgroundRangeMarkers.clear();
   }
   if (this.foregroundRangeMarkers!=null) {
     Set keys=this.foregroundRangeMarkers.keySet();
     Iterator iterator=keys.iterator();
     while (iterator.hasNext()) {
       Integer key=(Integer)iterator.next();
       clearRangeMarkers(key.intValue());
     }
     this.foregroundRangeMarkers.clear();
   }
   fireChangeEvent();
 }
```

```java
/* decompiled code */
public void clearRangeMarkers() {
  if (this.backgroundRangeMarkers!=null) {
    final Set keys=this.backgroundRangeMarkers
                                    .keySet();
    for (final Integer key:keys) {
      this.clearRangeMarkers(key);
    }
    this.backgroundRangeMarkers.clear();
  }
  if (this.foregroundRangeMarkers!=null) {
    final Set keys=this.foregroundRangeMarkers
                                    .keySet();
    for (final Integer key:keys) {
      this.clearRangeMarkers(key);
    }
    this.foregroundRangeMarkers.clear();
  }
  this.fireChangeEvent();
}
```

Fig. 4. Example of type-3 clones with different loops in *clearRangeMarkers()* that can be detected after decompilation

TABLE V
CHARACTERISTICS OF DISJOINT CLONES REPORTED IN $Cf_{orig\text{-}only}$ AND $Cf_{decomp\text{-}only}$

Clone set	Why are they not reported in another set?	JFreeChart			Tomcat			Total
		T1	T2	T3	T1	T2	T3	
$Cf_{orig\text{-}only}$	Too small after decompilation	1	9	32	1	6	120	169
	Too different after decompilation	0	6	11	21	0	0	38
	Smaller after decompilation causing higher dissimilarity	0	0	0	0	0	5	5
	Unknown	0	0	5	0	19	16	40
$Cf_{decomp\text{-}only}$	Having added/deleted statements, type casts, package names	3	5	8	2	0	1	19
	Having different `if-else` statements	0	12	3	0	0	0	15
	Using different loops (`for` vs. `while`)	0	0	4	0	0	0	4
	Inner-class methods	0	0	0	0	0	2	2
	Unknown	0	0	12	0	3	20	35

decompilation which provide several useful insights. In various cases, we find that decompiled clones are more compact and concise than the original code.

VI. CONCLUSIONS

Compilation/decompilation can be considered as a code normalisation method. It canonicalises several syntactic changes made to Java source code. We study compilation and decompilation as a pre-processing step for clone detection in three open source software systems. A clone mapping tool is utilised to map decompiled clone pairs back to their original locations. After the mapping, we compare and find common and disjoint clones before and after decompilation. The findings show that 78.7% of the clones are agreed before and after decompilation. By looking manually at 326 disjoint clone pairs (21.3%), we find that they are all true clone pairs except one. Hence, clone detection has perfect recall neither in the original nor in the decompiled version.

We summarise 7 characteristics of disjoint clones. More than half (67%) of the pairs in original-only set are pairs that become too small after decompilation. On the other hand, clones in the decompiled-only set are type-2 and type-3 pairs containing different statements, `if-else` statements, or loops. Some of them are discovered using even stricter type-2 configurations.

We plan to expand the study to a larger scale which covers more clone detectors, compilers, and software systems. Another direction is to compare Java bytecode and Android's Dalvik Executable format (dex). We can use a dex decompiler such as Androguard [8] to decompile dex to source code.

Lastly, we suggest to use decompilation as a complementary method to clone detection. We find that combining clones from before and after decompilation can increase recall without sacrificing precision.

REFERENCES

[1] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM'98*, pages 368–377, 1998.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *TSE*, 33(9):577–591, 2007.

[3] P. T. Breuer and J. P. Bowen. Decompilation: the enumeration of types and grammars. *ACM Transactions on Programming Languages and Systems*, 16(5):1613–1647, 1994.

[4] K. Chen, P. Liu, and Y. Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *ICSE'14*, 2014.

[5] C. Cifuentes and K. J. Gough. Decompilation of Binary Programs. *Software Practice and Experience*, 25(7):811–829, 1995.

[6] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage: Determining the provenance of software development artifacts. *Empirical Software Engineering*, 18:1195–1237, 2013.

[7] I. J. Davis and M. W. Godfrey. From Where It Came: Detecting Source Code Clones by Analyzing Assembler. In *WCRE'10*, pages 242–246, 2010.

[8] A. Desnos and G. Gueguen. Android: From reversing to decompilation. *Black Hat Abu Dhabi*, pages 1–24, 2011.

[9] D. M. German, M. Di Penta, Y.-G. Gueheneuc, and G. Antoniol. Code siblings: Technical and legal implications of copying code between applications. In *MSR'09*, pages 81–90, 2009.

[10] N. Göde and R. Koschke. Incremental clone detection. In *CSMR'09*, pages 219–228, 2009.

[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE'07*, 2007.

[12] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *TSE*, 28(7):654–670, July 2002.

[13] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS'01*, pages 40–56, 2001.

[14] O. Kononenko, C. Zhang, and M. W. Godfrey. Compiling Clones: What Happens? In *ICSME'14*, pages 481–485, 2014.

[15] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE'01*, pages 301–309, 2001.

[16] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *TSE*, 32(3):176–192, 2006.

[17] A. Mycroft. Type-Based Decompilation (or Program Reconstruction via Type Reconstruction). *Programming Languages and Systems*, 1999.

[18] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[19] Procyon java decompiler. https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler, Accessed: 2015-08-27.

[20] T. A. Proebsting and S. A. Watterson. Krakatoa: Decompilation in Java (does bytecode reveal source?). In *USENIX*, pages 185–198, 1997.

[21] C. Ragkhitwetsagul, J. Krinke, and D. Clark. Similarity of Source Code in the Presence of Pervasive Modifications. In *SCAM'16*, 2016.

[22] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC'08*, pages 172–181, 2008.

[23] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. SourcererCC: Scaling Code Clone Detection to Big-Code. In *ICSE '16*, pages 1157–1168, 2016.

[24] G. M. Selim, K. C. Foo, and Y. Zou. Enhancing Source-Based Clone Detection Using Intermediate Representation. In *WCRE'10*, pages 227–236, 2010.

[25] J. Svajlenko, I. Keivanloo, and C. K. Roy. Big data clone detection using classical detectors: an exploratory study. *Journal of Software: Evolution and Process*, 2014.

[26] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with BigCloneBench. In *ICSME'15*, pages 131–140, 2015.

[27] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *FSE'13*, pages 455–465, 2013.