# XIAO: Tuning Code Clones at Hands of Engineers in Practice

Yingnong Dang[1], Dongmei Zhang[1], Song Ge[1], Chengyun Chu[2], Yingjun Qiu[3*], Tao Xie[4]
[1]Microsoft Research Asia, China
[2]Microsoft Corporation, USA
[3]Alibaba Corporation, China, [4]NC State University, USA
{yidang;dongmeiz;songge;chchu}@microsoft.com, soloqyj@msn.com, xie@csc.ncsu.edu

## ABSTRACT
During software development, engineers often reuse a code fragment via copy-and-paste with or without modifications or adaptations. Such practices lead to a number of the same or similar code fragments spreading within one or many large codebases. Detecting code clones has been shown to be useful towards security such as detection of similar security bugs and, more generally, quality improvement such as refactoring of code clones. A large number of academic research projects have been carried out on empirical studies or tool supports for detecting code clones. In this paper, we report our experiences of carrying out successful technology transfer of our new approach of code-clone detection, called XIAO. XIAO has been integrated into Microsoft Visual Studio 2012, to be benefiting a huge number of developers in industry. The main success factors of XIAO include its high tunability, scalability, compatibility, and explorability. Based on substantial industrial experiences, we present the XIAO approach with emphasis on these success factors of XIAO. We also present empirical results on applying XIAO on real scenarios within Microsoft for the tasks of security-bug detection and refactoring.

## Categories and Subject Descriptors
D.2.7 [Software Engineering]: [Distribution, Maintenance, Enhancement]

## General Terms
Security, Algorithm

## Keywords
Code clone, code duplication, duplicated security vulnerability, code-clone detection, code-clone search

## 1. INTRODUCTION
During software development, engineers often reuse a code fragment via copy-and-paste with or without modifications or adaptations. Such practices lead to a number of the same or similar code fragments called code clones spreading within one or many large codebases. Detecting code clones [6][10][14][18][20] has been commonly shown to be useful towards various software-engineering tasks such as bug detection and refactoring.

In general, there are four main types of code clones [6][20]. Type-I clones are identical code fragments except for variations in whitespace, layout, or comments. Type-II clones are syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout, or comments. Type-III clones are copied fragments with further modifications such as changed, added, or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout, or comments. Type-IV clones are code fragments that perform similar functionality but are implemented by different syntactic variants.

Among these four types of code clones, type-III code clones with or without disordered statements, called *near-miss code clones*, are of high practical interest because they may potentially have a negative impact on the code quality and increase maintenance cost [10]. For example, problems might occur when some code is changed for fixing a bug but the same fix is not applied to its clones. Another example is inconsistent evolution of code clones, e.g., one piece of code is changed for supporting more data types, but its clones are not changed accordingly. Figure 1 shows an example near-miss clone (which indicates a bug) reported by a Microsoft engineer. The difference between the code snippets A and B is relatively large: one statement in the code snippet B (Line 16) is replaced by 4 statements in code snippet A (Lines 16-19), and the "if" statement in code snippet B (Lines 23-25) is updated as Lines 24-28 in A with significant changes in the "if" condition.

A large number of academic research projects [20] have been carried out on empirical studies or tool supports for detecting code clones. However, in practice, so far few such research projects have resulted in substantial industry adoption beyond the empirical studies conducted by researchers themselves. Although a few integrated development environments have integrated the generic feature of code-clone detection, this feature has limited support for real use in practice, and no industrial experiences are reported on the application of such feature.

In this paper, we attempt to address this issue and share to the community with experiences of carrying out successful technology transfer of our new approach of code-clone detection [8], called XIAO. XIAO has already been used by a large number of Microsoft engineers in their routine development work, especially engineers from a security-engineering team at Microsoft who have been using XIAO's online clone-search service since May 2009 to help with their investigation on security bugs. XIAO has been integrated into Microsoft Visual Studio 2012, to be benefiting a huge number of engineers in industry.

Based on our experiences [8] of collaborating with Microsoft engineers on using and improving XIAO along with our

---

* This work was done when this author worked for Microsoft Research Asia.



*ACSAC '12* Dec. 3-7, 2012, Orlando, Florida USA

```
    // 3 identical statements omitted here
4.  switch (biBitCount)
5.  {
    // 9 identical statements omitted here
15. case 24: // 24bpp: Read colours from pixel
16. case 32:
17. palEntry.rgbRed    = ((RGBQUAD *)pPixel)->rgbRed;
18. palEntry.rgbGreen = ((RGBQUAD *)pPixel)->rgbGreen;
19. palEntry.rgbBlue    = ((RGBQUAD *)pPixel)->rgbBlue;
20. break;
21. default: // What else could it be?
22. return 0;
23. }
24. if (palEntry.rgbRed >= 0xFE && palEntry.rgbGreen >= 0xFE &&
25. palEntry.rgbBlue >= 0xFE ||((palEntry.rgbRed >= 0xbf &&
26. palEntry.rgbGreen >= 0xbf && palEntry.rgbBlue >= 0xbf) &&
27. (palEntry.rgbRed <= 0xc1 && palEntry.rgbGreen <= 0xc1 &&
28. palEntry.rgbBlue <= 0xc1)))
29.     return FALSE;
30. return TRUE;
```

**Code Snippet A**

```
    // 3 identical statements omitted here
4.  switch (biBitCount)
5.  {
    // 9 identical statements omitted here
15. case 24: // 24bpp: Read colours from pixel
16.     palEntry = *(RGBQUAD *)pPixel;
17.       break;
18.
19.
20.      default:    // What else could it be?
21.      return 0;
22.    }
23.   if (palEntry.rgbRed == 0xFF && palEntry.rgbGreen ==
    0xFF
24.     &&palEntry.rgbBlue == 0xFF || palEntry.rgbRed == 0xC0
25.     palEntry.rgbGreen == 0xC0 && palEntry.rgbBlue == 0xC0)
26.
27.       return FALSE;
28.    return TRUE;
```

**Code Snippet B**

**Figure 1. An example of near-miss code clones in a commercial codebase**

observations on real use of XIAO by Microsoft engineers, we attribute the success of XIAO to four main factors: its high tunability, scalability, compatibility, and explorability.

**High tunability** of XIAO is achieved with a new set of similarity metrics in XIAO, reflecting What You Tune Is What You Get (WYTIWYG): users can intuitively relate tool-parameter values with the tool outputs, and easily tune tool-parameter values to produce what the users want. For example, the similarity-parameter value of 100% should lead to outputs of two exactly same cloned snippets, and the 80% value should lead to outputs of two cloned snippets with 80% similarity judged by the users. The parameters of the proposed metrics in XIAO enable users to effectively control the degree of the syntactic difference between the two code snippets of a near-miss clone pair: the degree of the statement similarity, the percentage of inserted/deleted/modified statements in the clone pair, the balance between the code-structure similarity, and the quantity of disordered statements. Such high tunability of XIAO is critical in applying an approach of code-clone detection such as XIAO to a broad scope of software-engineering tasks such as refactoring and bug detection since these different tasks would require different levels of parameter values.

**High scalability** of XIAO in analyzing enormous lines of code is achieved with a well-designed scalable and parallelizable algorithm with four steps. These four steps include preprocessing, coarse matching, fine matching, and pruning. Preprocessing transforms source-code information to filter out inessential information such as code comments, and map code entities such as keywords and identifiers to tokens. Such information preprocessing reduces the cost burden of the actual analysis. To offer high scalability, XIAO splits the main analysis into two steps: coarse matching and fine matching. Coarse matching is less costly but less accurate than fine matching. The scope narrowed down by coarse matching is fed to fine matching, achieving a good balance on analysis scalability and accuracy. The step of pruning further improves the analysis accuracy. In addition, the clone-detection algorithm of XIAO can be easily parallelized. XIAO partitions the codebase and performs code-clone detection on each code-partition pair. Each instance of XIAO detects clones on a number of pairs. The results of all the instances are then merged.

**High compatibility** of XIAO in analyzing code in different development environments (such as different build systems) is achieved with its compiler-independent lightweight and pluggable parsers. XIAO has built-in parsers for the C/C++ and C# languages. We define an open Application Programming Interface that allows the easy plug-in of parsers to support various programming languages. It should be noted that the parsing task is lighter than the comprehensive functionalities offered by compilers. Compared with approaches of parse-tree-based clone detection such as Deckard [14][9], our approach has the advantage of compiler independence; it can be easily applied to accommodate different language variants and build environments, which typically exist in real settings of software development, especially for C/C++ [7].

**High explorability** of XIAO in supporting users to easily explore and manipulate detected code clones is achieved with its well-designed user interfaces including visualization support. We design a simple heuristic to define the level of difference between cloned snippets. We also use the metric to rank clones to prioritize the review of clones to identify bugs. XIAO includes clone visualization to clearly show the matching blocks and the block types of a clone pair. This way, users can quickly capture whether there is any difference between the two cloned snippets, what kind of difference it is, and how much difference there is. XIAO also includes a tagging mechanism to help coordinate joint efforts of reviewing code clones from multiple engineers.

We have released XIAO to Microsoft engineers since April 2009 and a great number of Microsoft engineers from different teams have used it. XIAO has been integrated into Microsoft Visual Studio 2012, to be benefiting a huge number of engineers in industry.

The rest of this paper is organized as follows. We present our code-similarity metric in Section 2. We introduce our clone-detection algorithm and visualization/reporting in Section 3 and Section 4, respectively. We present our empirical study in Section 5 and report several real-use scenarios in Section 6. Section 7 discusses related work and Section 8 concludes.

## 2. CODE SIMILARITY METRIC

Considering possible edits that can be applied to source code after it has been copied and pasted, we have identified three important

inconsistencies that are present in near-miss clones and that should be measured:

1. Statement-level difference, which may be caused by adapting the copied code to the coding style of the source file where it is copied to, e.g., a few identifiers are renamed;
2. Inserted/deleted/modified statements, which can be the result of changes to the code necessary for fixing bugs or implementing new features;
3. Disordered statements, which may be related to either a change of code logic or coding style.

We define our code-similarity metric to take these three types of inconsistencies into account. Before we define our metric, we next define three binary relationships between source-code statements. Let $\sum$ be an alphabet whose symbols are syntactical tokens, and $s \in \sum^*$ be a source-code statement.

**Definition 1 (Exact-Match)** Let $s_1$ and $s_2$ be two source-code statements; let $|s|$ be the token length of the statement *s*, and *s[i]* be the *i*-th token of *s*. Then $s_1$ and $s_2$ are *Exact-Match*-related if and only if $|s_1| = |s_2|, and\ s_1[i] = s_2[i],\ 0 < i \leq |s_1|$.

**Definition 2 (Transformed-Match)** Let $T: \sum \rightarrow \sum$ be a token-mapping function (e.g., mapping several different identifiers to the same token). The statements $s_1$ and $s_2$ are *Transformed-Match*-related if and only if $|s_1| = |s_2|, and\ T(s_1[i]) = T(s_2[i]),\ 0 < i \leq |s_1|$.

**Definition 3 (α-Transformed-Match)** The statements $s_1$ and $s_2$ are *α-Transformed-Match*-related if and only if they are *Transformed-Match*-related and there exist at least $\alpha * |s_1|$ (denoted as $l$) distinct indexes $i_1, \ldots, i_l$ such that $s_1[i_k] = s_2[i_k], 0 < k \leq l$.

Intuitively, two statements are *Exact-Match*-related if they are identical after code formatting is ignored, and are *Transformed-Match*-related if they are identical after both code formatting and identifier renaming are ignored. The *α-Transformed-Match* relationship provides the flexibility for controlling the degree on what percentage of renamed identifiers between two matched statements are tolerated. In fact, the *α-Transformed-Match* relationship is equal to the *Transformed-Match* relationship when $\alpha = 0$, and equal to the *Exact-Match* relationship when $\alpha = 1$.

We next illustrate these three relationships with examples. Consider that the parameterized versions of three statements A, B, and C listed below are identical.

| Original statements | Parameterized statements |
|---|---|
| A: If (foo(a, b, c) == null) | A1: If ( T ( T, T, T) == T) |
| B: If (foo(a, b1,c1) == null) | B1: If ( T ( T, T, T) == T) |
| C: If (bar(x, y, z) == null) | C1: If ( T ( T, T, T) == T) |

Consequently, any two of them are *Transformed-Match*-related. Each statement has 13 tokens. Statements A and B have 11 identical tokens out of the 13 tokens (84.6%); Statements A and C have 9 identical tokens out of the 13 tokens (69.2%). If $\alpha = 0.6$, A is *α-Transformed-Match*-related to both B and C; if $\alpha = 0.8$, A is *α-Transformed-Match*-related to only B.

In practice, Statements A and C could have a totally different semantic and could not be caused by copy-and-paste, thus being of low interest to users who focus on copy-and-paste clones.

**Definition 4 (Disordered-Match-Score, in short as DMS)** Consider two code snippets $S_1$ and $S_2$ where *m* distinct statements in $S_1$ at positions $i_1 < i_2 < \cdots < i_m$ match *m* distinct statements at positions $i'_1, i'_2, \cdots, i'_m$ in $S_2$. Let *d* be the number of inversion pairs[1] in the index sequence $\{i'_1, i'_2 \cdots i'_m\}$. The *Disordered-Match-Score (DMS)* of $S_1$ and $S_2$ is defined as

$$DMS(S_1, S_2) = \frac{2*d}{m-1} \quad (1)$$

The *DMS* measures the structure difference of the two code snippets. The smaller the value of the *DMS* is, the more similar the structures of the two code snippets are because there are fewer disordered statements. The possible value of the *DMS* is from 0 (when there is no inversion pair) to *m* (when the order is fully reversed). Figure 2 shows an example of two code snippets with disordered statements: the statement in Line 4 on the left snippet is moved to Line 2 on the right. The statements in Lines 1 to 5 on the left snippet correspond to the statements in Lines 1, 3, 4, 2, and 5 on the right snippet. The inversion pairs of this index sequence are (3, 2) and (4, 2). The *DMS* of the two snippets in the example is 1 (being 2*2/4) .

```
1  for (i = 0; i < 10; i++) {        1  for (i = 0; i < 10; i++) {
2     a++;                           2     c=foo(a, b);
3     b++;                           3     a++;
4     c=foo(a, b);                   4     b++;
5     d=bar(a, b, c);}               5     d=bar(a, b, c);}
```

**Figure 2. Example of disordered statements**

**Definition 5 (α-Transformed Similarity)** Let $|S|$ be the number of statements of a code snippet *S*. The *α-Transformed Similarity* between two snippets $S_1$ and $S_2$ is defined as

$$\alpha Sim(S_1, S_2) = 2\frac{m - \theta DMS(S_1, S_2)}{|S_1| + |S_2|} \quad (2)$$

where m is as in Definition 4 and $\theta$ is a penalty coefficient to the DMS. We use the α-Transformed similarity to measure code similarity hereafter.

**Definition 6 (Transformed Similarity)** The *Transformed Similarity*, $TSim(S_1, S_2)$ is defined as its *α-Transformed Similarity* with $\alpha = \theta = 0$.

**Definition 7 (Clone Pair)** Code snippets $S_1$ and $S_2$ are a clone pair when $\alpha Sim(S_1, S_2) \geq \gamma, 1 \geq \gamma > 0$, where γ is a configurable similarity threshold.

Definition 7 satisfies the requirement posed at the beginning of this section. It controls the statement-level difference by the value of α. It controls the number of inserted/deleted statements proportionally to the size of the code snippets by the value of γ; the bigger the snippets, the more inserted/deleted statements are tolerated. The penalty coefficient θ permits users to control the amount of disordered statements in a clone pair.

# 3. CLONE-DETECTION ALGORITHM

Figure 3 shows an overview of our clone-detection algorithm, consisting of four steps: preprocessing, coarse matching, fine matching, and pruning. Our algorithm takes one codebase as input and produces code clones detected from the codebase (our

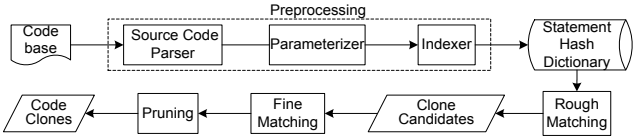


**Figure 3. Overview of XIAO's algorithm of code-clone detection**

[1] http://en.wiktionary.org/wiki/inversion_pair

algorithm can handle multiple codebases by treating them as one codebase).

In the preprocessing step, the source-code parser extracts the location information of all the functions and their statements. Then the code is parameterized and indexed similar to the preprocessing techniques of CP-Miner [21].

In the coarse-matching step, for each function f in the codebase, a list of its clone-candidate functions $\{\mathrm{CF_f}\}$ is detected. Each candidate function has a sufficient number of statements with the same hash value as at least one statement in f. This step helps reduce the search space of the fine-matching step.

In the fine-matching step, we identify all clone pairs between each function f and each of its clone-candidate functions $\{\mathrm{CF_f}\}$ using the metric in Definition 6, with $\alpha = \theta = 0$. The setting of $\alpha = 0$ enables us to use the hash values of the parameterized statements to easily verify the matching relationship; $\theta = 0$ enables us to easily calculate the similarity using Equation (2).

In the pruning step, we recalculate the similarity of the clone pairs obtained in the fine-matching step, using the user-specified non-zero values of α and θ, and thus prune the clone pairs with a similarity that is less than the similarity threshold γ. We next give the details of the last three steps in this section.

## 3.1 Coarse Matching

Given a function f and a statement-hash dictionary D, the coarse-matching algorithm returns a list of candidate functions $\{\mathrm{CF_f}\}$ so that at least a minimum and sufficient number of statements in f and any function in $\{\mathrm{CF_f}\}$ have the same hash values. Doing so ensures that only functions sharing a minimum and sufficient number of statements are searched for code clones. In this way, the search space is reduced from the whole input codebase to just $\{\mathrm{CF_f}\}$. All possible function pairs that potentially contain cloned code snippets are identified by performing the coarse matching on all the functions in the input codebase. The steps of fine matching and pruning are then performed between f and each function in $\{\mathrm{CF_f}\}$ to obtain actual clones.

We next define the concepts of the Hit Function and Clone Candidate Function to help illustrate the coarse-matching algorithm.

**Definition 8 (Hit Function)** Let $H: \sum^* \to \mathbb{N}$ be a hash function, and $T^*: \sum^* \to \sum^*$ be the extension of the token-mapping function $T$ (see Definition 2) to whole statements. A function $F_{hit}$ is named as a Hit Function of a function $f$ if there exist a statement $s$ in $f$ and a statement $s_h$ in $F_{hit}$ that satisfy $H(T^*(s))) = H(T^*(s_h)))$.

**Definition 9 (Clone-Candidate Function)** A function $CF_f$ is a Clone-Candidate Function of a function $f$ if there exist at least $n_{match}$ statements in $f$ with $CF_f$ as one of its Hit Functions and

$$n_{match} \geq \min\left(\frac{\gamma}{2-\gamma} \cdot L, \quad \gamma \cdot MinS\right) \qquad (3)$$

where $L$ is the number of the statements in $f$, $\gamma$ is the clone-similarity threshold in Definition 7, and $MinS$ is the minimal number of statements that a cloned snippet should have.

Intuitively, a Hit Function $F_{hit}$ has at least one parameterized statement in common with function $f$. $CF_f$ has at least $n_{match}$ common parameterized statements with $f$.

Suppose that $D$ is the hash dictionary of an input codebase. For every statement $s$ in function $f$, the coarse-matching algorithm uses $D$ to generate a list of Hit Functions $\{F_{hit}\}$ by retrieving functions each of which contains a parameterized statement with the same hash value as that of the parameterized form of $s$. $\uplus Hit(f)$ is the multiset union of all the functions in the Hit Function lists for every statement of $f$. The total hit count of each function in $\uplus Hit(f)$ is equal to the function's multiplicity in $\uplus Hit(f)$. We then identify a list of Clone-Candidate Functions of $f$ as those functions in $\uplus \mathrm{Hit}(f)$ with no less than $n_{match}$ occurrences.

## 3.2 Fine Matching

The coarse matching identifies a list of Clone-Candidate Functions for each function $f$ in the input codebase. There may not be clone pairs between $f$ and $CF_f$ for the following reasons:

(a) the matched parameterized statements may be so scattered in $f$ and $CF_f$ that the similarity between the snippets in $f$ and $CF_f$ is not high enough;
(b) multiple parameterized statements in $f$ or $CF_f$ may be mapped to the same tokenized statement in $CF_f$ or $f$, causing that the number of one-to-one matched statements between $f$ and $CF_f$ is not high enough;
(c) two statements are not necessarily α-*Transformed-Match*-related even if they have the same hash value;
(d) there might be mismatched statements between $f$ and $CF_f$ due to hash collisions, although the probability of hash collision is quite low;
(e) some matched statements could be instances of disordered matches and the penalty to the disordered match in Equation (2) would cause that the similarity is not high enough.

We address issues (a) and (b) in the fine-matching step and issues (c), (d), and (e) in the pruning step.

The goal of the fine-matching step is to identify all snippet pairs (between $f$ and $CF_f$) whose Transformed Similarity (Definition 6) is not less than a specified threshold. We formulate this problem as finding code snippets $S_1$ and $S_2$ in $f$ and $CF_f$, respectively, that satisfy

$$\left.\begin{array}{ll} H\left(T^*(s_{1,i})\right) = H\left(T^*(s_{2,i})\right) & (a) \\ \frac{2m}{|S_1|+|S_2|} \geq \gamma & (b) \end{array}\right\} \qquad (4)$$

where $s_{1,i}$ and $s_{2,i}$, ($i$ = 1, … , $m$) are $m$ statements in $S_1$ and $S_2$, respectively. Equation (4.a) ensures that there are $m$ matching parameterized statements; Equation (4.b) ensures that the $\alpha$-*Transformed Similarity* of $S_1$ and $S_2$ is not less than the similarity threshold $\gamma$ given the values of $\alpha$ and $\theta$ in Equation (2) are equal to 0.

We next first present how to determine whether a given snippet pair $S_1$ and $S_2$ satisfies equation (4), and then present how to efficiently scan $f$ and $CF_f$ to find all the possible pairs of $S_1$ and $S_2$ in $f$ and $CF_f$.

To determine whether $S_1$ and $S_2$ satisfy Equation (4), we calculate the value of $m$ as follows. Suppose that (1) $\{V_i | i = 1,2,\cdots,t\}$ is the list of the hash values for which at least one statement in $S_1$ and one statement in $S_2$ are mapped to $V_i$, and (2) there are also $n_{1,i}$ and $n_{2,i}$ statements with the hash value $V_i$ in $S_1$ and $S_2$, respectively. It easily follows that there are $n_i = \min(n_{1,i}, n_{2,i})$ matched parameterized statements in $S_1$ and $S_2$. Therefore, $m$ can be easily calculated as

$$m = \sum_{i=1}^{t} n_i \qquad (5)$$

Accordingly, we determine whether $S_1$ and $S_2$ satisfy Equation (4).

The next subtask is to scan all the possible snippet pairs in $f$ and $CF_f$. We take a two-step procedure. First, given a snippet $S_1$ in $f$,

we scan all the possible $S_2$ in $CF_f$ and determine whether $S_1$ and $S_2$ satisfy Equation (4). Second, we enumerate all the possible $S_1$ in *f* and repeat the first step.

During the first step, we use a sliding window on top of the statement sequence of $CF_f$ to enumerate all the code snippets in $CF_f$. The statement sequence inside the window is the current code snippet $S_2$. To satisfy Equation (4), the number of statements of $S_2$ in $CF_f$ should satisfy the following constraint:

$$k_{min} \le |S_2| \le k_{max} \quad (6)$$

where $k_{min} = \frac{\gamma}{2-\gamma}|S_1|$, $k_{max} = \frac{2-\gamma}{\gamma}|S_1|$. Therefore, we need to use a set of sliding windows with sizes ranging from $k_{max}$ to $k_{min}$ to enumerate all possible snippets in $CF_f$. Given a sliding window size *k*, the window starts from position 1 ($W_1$) that covers the first *k* statements in $CF_f$. After checking whether the snippet inside the window and $S_1$ satisfy Equation (4), the window moves one step further to position 2 ($W_2$), and so on. Compared with the code snippet covered by $W_1$, the code snippet covered by $W_2$ has only the first statement of $CF_f$ removed and the statement in position *k*+1 added. Therefore, we calculate the value of *m* for the code snippet in $W_2$ by just updating the value of *m* for the code snippet in $W_1$, i.e., by removing the contribution of the first statement and adding the contribution of the added statement in Equation (5).

During the second step, we use a sliding window to enumerate all the possible snippets $S_1$ in *f*, and repeat the first step. The size of this sliding window ranges from $|CF_f|$ (the total number of statements in $CF_f$) to *MinS* (the minimal number of statements that a cloned snippet should have).

We further optimize the algorithm in a number of ways. For example, the sliding windows in the first step could directly move to the next statement that matches at least one statement in *f*. In addition, once a snippet pair is identified as passing the fine matching, we further execute the pruning step against the pair to determine whether it is an actual clone pair or not. Once a snippet pair passes the pruning, we continue to perform the fine matching in the remaining parts of *f* and $CF_f$; in this way we avoid getting overlapped clone pairs.

## 3.3 Pruning

In the pruning step, we prune the snippet pairs obtained in the fine-matching step to get code clones that satisfy our code-clone definition with the specified non-zero values for $\alpha$ and $\theta$ in Equation (2). This step addresses issues (c), (d) and (e) mentioned at the beginning of Section 3.2.

To address these three issues, we need to get the *α-Transformed-Match*-related statements (Definition 2) in the two code snippets in the pair such that the *Disordered-Match-Score* (DMS) (Definition 4) of the two snippets is minimized. We then calculate the *α-Transformed-Similarity* based on Equation (2) and discard the snippet pair if its *α-Transformed-Similarity* value is lower than the threshold.

We use a greedy technique called Karp-Rabin Matching and Greedy String Tiling [30] to get the matched statements. The basic idea is to use a dynamic-programming algorithm to find the maximal consecutive statement sub-sequences $S^{1,1}$ in $S_1$, and $S^{2,1}$ in $S_2$, with the same number of statements, and each statement in $S^{1,1}$ *α-Transformed-Match*-related with the statement at the corresponding position in $S^{2,1}$. The next step is to exclude the statements in $S^{1,1}$ and $S^{2,1}$ from $S_1$ and $S_2$, respectively, and repeat the step on $S_1 \backslash S^{1,1}$ and $S_2 \backslash S^{2,1}$. By reiterating this process until there are no further matches, we get a set of statement-sub-sequence pairs in $S_1$ and $S_2$, which are *α-Transformed-Match-related* to each other. The matched statements that we need to obtain are the union of all the sub-sequence pairs. At this point, we calculate the *α-Transformed Similarity* and determine whether $S_1$ and $S_2$ are a clone pair based on Definition 7.

# 4. VISUALIZATION AND REPORTING

As important and integral components of XIAO, the clone visualization and reporting mechanism provides a rich and interactive user experience for engineers to efficiently review the clone-analysis results and take corresponding actions.

**Clone reporting.** We design a simple heuristic to define the level of difference between cloned snippets. In particular, it first filters out all those exactly the same cloned snippets, since cloned snippets with slightly different logics would be more bug-prone. We use a metric (called bug likelihood) to rank clones to prioritize the review of clones to identify bugs. We also design a simple heuristic to measure in what extent the cloned snippets are similar to each other and how easily they can be refactored (e.g., the exact same copies could be easier to be refactored than others). We call this metric as refactoring likelihood. To facilitate users to act on the reported clones, we have developed XIAO's Clone Explorer, a component of clone reporting and exploration shown in Figure 4. It organizes clone statistics based on the directory hierarchy of source files in order to enable quick and easy review at different source levels (Figure 4①). A drop-down list (②) is provided to allow pivoting the clone-analysis results around the bug likelihood (③), refactoring likelihood, and clone scope. Clone scope indicates whether cloned snippets are detected inside a file, cross-file, or cross-folder. For a selected folder in the left pane, the right pane (④) displays the list of clone functions (those including cloned snippets), which could be sorted based on bug likelihood or refactoring likelihood (⑥). Filters (⑤) on the clone scope, bug likelihood, or refactoring likelihood are provided to enable easy selection of clones of interest.

**Clone visualization.** Figure 5 shows how the Clone-Visualizer component visualizes the clone pair illustrated in Figure 1. The key to clone visualization is to clearly show the matched statement blocks and the block types. We categorize the matched blocks into the following types: exactly same (i.e., there are only possible formatting differences), similar-logic block (i.e., there are identifier substitutions between the two blocks), different logic (i.e., the statements in the two blocks are not of the similar-logic type but are still similar), and extra logic (i.e., the statements of a block show up in one copy of the clone pair, but not in the other copy). In this way, users can quickly determine whether there is any difference between the two cloned snippets, what kind of difference it is, and how much difference there is. Blocks are numbered for correspondence display (Figure 5 ①), and different colorings are used to indicate different block types (②). The left and right source panes are synchronized, and navigation buttons are provided to navigate through source code by matched blocks instead of statements in order to improve review efficiency (③). Users can take an immediate action of filing a bug once a clone is confirmed to be a bug or a refactoring target (⑤), or copying the code out for more investigation (④).

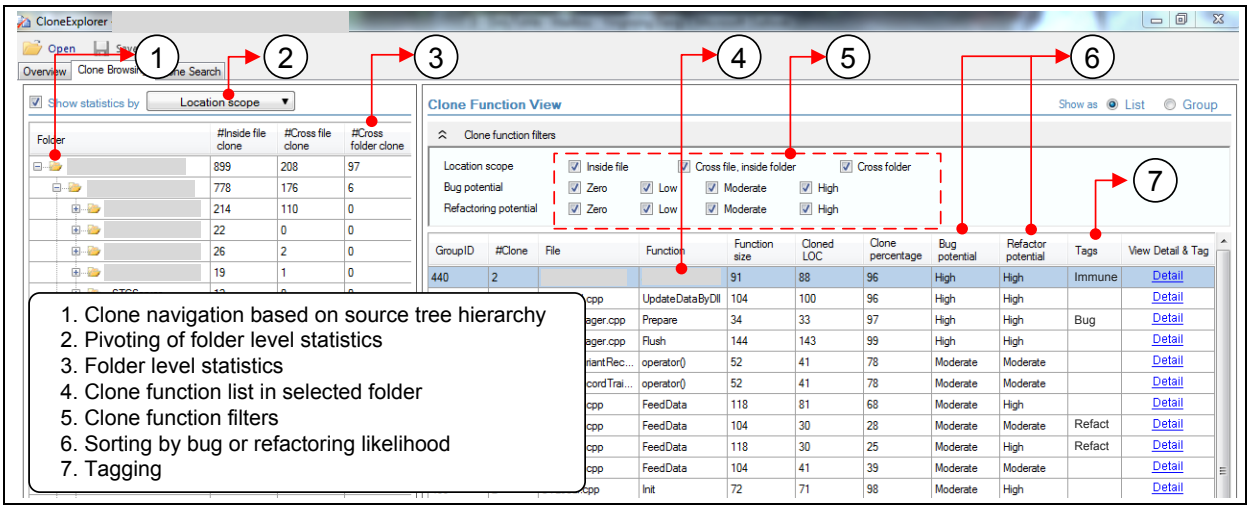


**Figure 4. UI of Clone Explorer**


**Figure 5. Visualizing differences of a clone pair**

**Tagging.** One important requirement of XIAO is to help coordinate joint efforts of reviewing code clones from multiple engineers. We have designed a tagging mechanism for engineers to easily work together. One clone already reviewed by an engineer can be tagged as "immune"[2], "bug", or "refactoring". Then the other reviewing engineers could choose to easily skip these already reviewed clones. Note that these tags need to be tracked as done in XIAO when a new version of codebase is analyzed. Overall, a tagging mechanism (Figure 4 ⑦) serves two main purposes. First, users can tag some clones as "immune" at various occasions. For example, some detected clones do not include buggy code or become refactoring targets. Second, we can implicitly collect user feedback and evaluation results in order to keep improving our clone-analysis algorithms.

# 5. EMPIRICAL STUDIES

In this section, we present the empirical results of applying XIAO on commercial codebases. In our studies, we used seven commercial codebases at Microsoft. In the seven commercial codebases, six are in C/C++ and one is in C#; the numbers of lines of code vary between 1.9 million and 12 millions.

The environment for running XIAO was a workstation running Windows 7 64 bits with two Intel Xeon 2.0GHz processors and 12GB memory. We relied on human inspection to classify whether a detected clone is a real clone.

## 5.1 Clone-Detection Effectiveness

Figure 6 shows the distribution of the types of code clones detected by XIAO across the seven commercial codebases, when using the default settings: *MinS* = 10, α = 0.6, γ = 0.8. The figure shows that the near-miss clone pairs detected by XIAO are a significant portion of all the clone pairs, ranging from 63% to 93% for the commercial codebases.

On each of two commercial codebases (out of the seven) at Microsoft, one of its Microsoft engineers (i.e., those who developed the codebase and are familiar with the codebase) helped evaluate some clone-analysis results generated by XIAO on the codebase. We named these two engineers as Engineers I and II.

[2] An immune clone is one of no particular interest to engineers.

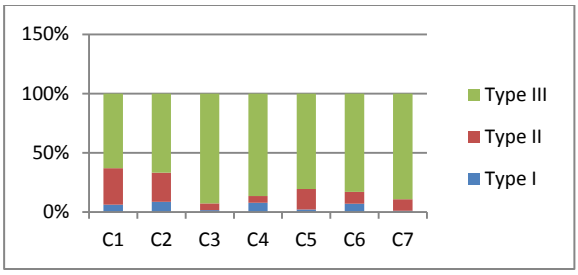


**Figure 6. Distribution of clone types of seven commercial codebases (all in C/C++ except C1 in C#) detected by XIAO**

Engineer I reviewed 69 clone groups (each of which includes a set of similar clone pairs) with 184 functions in total. All reviewed

```
    // 14 identical statements omitted here
    ::SendMessage(hwndCombo, CB_LIMITTEXT, GetMaxCharacters(),
0);
    int iFlags = 0;
    if (!GetIsIMEAvailable())
    iFlags |= SES_NOIME;
    if (iFlags)
    ::SendMessage(hwndCombo, EM_SETEDITSTYLE, iFlags, iFlags);
    // 5 identical statements omitted here
    ::SendMessage(hwndCombo, EM_SETCOMBOBOXSTYLE,
    SCB_NOAUTOCOMPLETEONSIZE, SCB_NOAUTOCOMPLETEONSIZE);
    // 2 identical statements omitted here
```

**Figure 7. A confirmed bug: extra statements for bug fixing were added (with the gray background) to one function but not to its cloned one.**

functions are of non-zero bug likelihood and refactoring likelihood. Using the tagging functionality of XIAO, Engineer I tagged 7 (10%) clone groups as potential bugs and 16 (23%) clone groups as refactoring targets. All together there were 23 (33%) clone groups that were identified as actionable (i.e., either potential bugs or refactoring targets).

Engineer II evaluated a small set of clones found by XIAO in a system component that consists of high-quality source code. The source code of this component has been stable with few changes for a number of years. We did not expect to find clone-related bugs in this case. Instead, we were interested in looking for refactoring targets in high-quality code. Engineer II reviewed a total of 39 clone groups with 102 functions. The numbers of clones in these clone groups vary from 2 to 7, except one clone group, which contains 20 clones. All these 20 clones deal with Windows Event operations and they have slight differences in code logic. Including this clone group, Engineer II tagged 8 (16.3%) clone groups with 46 functions as refactoring targets.

## 5.2 Runtime Cost and Scalability

The running time of XIAO against a large codebase (with the default environment) varies depending on the used settings: from 6 minutes (*MinS* = 20, $\alpha = 1$, $\gamma = 1$) to 23 minutes (*MinS* = 10, $\alpha = 0.4$, $\gamma = 0.8$). Basically, increasing $\gamma$ tends to linearly decrease the spent time; increasing *MinS* decreases the spent time; increasing $\alpha$ does not change the spent time. This behavior can be easily explained: increasing the value of $\gamma$ leads to a smaller number of clone-candidate functions in the coarse-matching step, thus decreasing the time spent in each of the successive steps; increasing *MinS* leads to a smaller number of snippets to be checked; $\alpha$ is used in the pruning step, which is the last step, and affects only the number of obtained clones but does not affect the spent time.

Instead of using the default environment, we evaluated the scalability of our XIAO system using an HPC cluster with one master node and four computing nodes (a high-performance computing environment that XIAO leverages to deal with a huge number of lines of code). The master node has four AMD Opteron 880 Dual-core 2.4GHz CPUs and 32GB memory. Each of the four computing nodes has two Intel E5335 Dual-core 2.0GHz CPUs and 8GB memory. Both the master and computing nodes are running on Windows Server 2008 HPC Edition.

**Online clone search.** We indexed a commercial codebase with about 130 million lines of code to evaluate the scalability of XIAO's online clone-search engine. Code snippets with three or more statements are accepted as valid input for clone search. The preprocessing (including source-code parsing, tokenization, and indexing) was conducted on one computing node and it took 3 hours and 42 minutes to finish. Source code is divided into partitions each with 5MB storage size and these partitions are evenly distributed on the four computing nodes. Then 16 instances of the online clone-search engine were running to serve online queries. We randomly selected 1000 code snippets from the codebase as inputs. The size of these 1000 snippets ranges from 3 to 100 and the number of snippets for each size is about the same. The clone-similarity threshold is set to be 0.6. The number of found cloned snippets ranges from 1 to 1000 and the average time of each query is a number of seconds.

**Offline clone detection and analysis.** We evaluated the performance of XIAO's offline clone detection and analysis on a commercial codebase with 26 million lines of code using the same system setup as that in the online-search environment. Clones of functions with at least 20 lines of statements were found using the similarity threshold of 0.6. Preprocessing was conducted on one computing node. The clone detection and analysis were performed in parallel on the 4 computing nodes. It took 3 hours and 30 minutes to finish the entire process. The time breakdown of each step (in the unit of seconds) is preprocessing (1,014), coarse matching (9,803), fine matching (213), and clone analysis (1,462). The average amount of memory used by each instance of clone detection and analysis is about 120MB.

# 6. APPLICATION SCENARIOS IN PRACTICE

We have released XIAO inside Microsoft for different development teams to use (with the first version released in April 2009). There were more than 750 downloads of the tool as of the end of year 2010.

**Copy-Paste-Bug Detection and Refactoring.** An example application scenario of XIAO was already described in Section 1. In this scenario, an engineer at Microsoft reviewed 69 clone groups for a total of 184 snippets taken from the results of code-clone detection for a commercial codebase. All reviewed clones were near-miss code clones. He identified 7 (10%) clone groups as potential bugs and 23 (33%) clone groups as refactoring targets (including the 7 with potential bugs). The motivating example shown in Figure 1 is one of these seven cases. Function A on the left side is from a shared component, and function B on the right side is from an application. As confirmed by the code owner, B was copied from A for quick reuse quite some time ago. However, the engineer of B was not aware of the changes made to A after the copying.

The clone-related bug shown in Figure 7 is another example reported by the same engineer. In this case, the two functions originally had similar functionalities. Later on a number of statements were added to one function (with the gray background in Figure 7) to ensure the synchronization between Windows GDI objects; nevertheless, this bug fix was not applied to the other function.

The two functions shown in Figure 8 have only slight differences. In fact, they are the same except for one similar-logic block (the second statement in the figure) and one different-logic block (the first statement). This case was analyzed by XIAO to have a high rank in both bug likelihood and refactoring likelihood. As confirmed by its engineer, the differences between the two functions are by-design, and the clones are not buggy. In the meantime, this case was confirmed to be refactorable.

Figure 9 shows a clone group that was tagged as "Immune". Although there do exist slight logic differences between the two functions, the differences were confirmed to be intentional. Currently it is difficult for XIAO to handle false-positive cases such as this one in clone analysis.

Based on our observation, an engineer often tries to prioritize his refactoring efforts, i.e., starting from easy-to-refactor clones (which are often those with high similarity). Another factor for tuning parameters is that a higher value of the similarity threshold needs less running time to get clone-detection results. Therefore, the engineer could choose relatively high similarity threshold first (e.g., 100% the same), to get some easy-to-refactor clones within relatively short clone-detection time. If there is a need to aggressively identify more refactoring opportunities, a relatively small value of the similarity threshold could be used. In some situations, a relatively high value of the similarity threshold would be used. For example, we observed that engineers dealing with a codebase with 20+ million LOC would like to identify file-level clones with 99% similarity and set a relatively high value of the similarity threshold to accomplish this goal.

**Detection of Duplicated Vulnerable Code.** A security-engineering team at Microsoft has been using XIAO's online clone-search service since May 2009 to help with their investigation on security bugs. There were more than 590 million lines of code being indexed. During the second half of year 2010, there were a number of vulnerable code snippets searched against the XIAO service. Among these searching cases, there were 18.3% cases with good hits, i.e., for these cases, the security-engineering team needs to do further investigation to confirm whether there are duplicated vulnerabilities. Given high severity of security bugs, 18.3% good-hit cases are very good results.

In an example real case, a reported security vulnerability could cause potential heap corruption and lead to remote code execution. After investigation, the vulnerable code snippet was found in codebase A: a buffer-overflow check was missing there.

Using XIAO's clone-search service, one security engineer on the security engineering team found three clones of the vulnerable code snippet – one is also in codebase A and the other two belong to codebase B. This security engineer contacted the code owners of these three cloned snippets and confirmed that one snippet in codebase B was vulnerable. After the contact, the development team owning the vulnerable cloned snippet in B had confirmed to fix this security bug while the security bug in codebase A was fixed.

XIAO's clone-search service has greatly improved the productivity of the security engineers and it enhanced the reliability of the bug-investigation process as well. Based on the clone-search results, security engineers are able to obtain a better understanding of the potential impact of security vulnerabilities and communicate more effectively with development teams on vulnerability investigation and fixing.

In this application scenario of XIAO, security engineers would like to have high recall of clone detection (i.e., little chance of missing clones). Therefore, for this application scenario, XIAO has the default value of 0.6, a relatively small value for the similarity threshold. The value is tunable by security engineers to achieve even higher recall.

**Discussion.** For the two types of application scenarios, we observed that the second scenario on detecting duplicated vulnerable code (with the target users as security engineers) has occurred much more often than the first scenario, especially on refactoring (with the target users as software engineers). Such observation could be explained with two factors. First, refactoring conducted by software engineers occurs much less frequently than investigation of security bugs, which are the routine work of security engineers. Second, the severity of consequence on missing a refactoring opportunity is much less than the one on missing a security bug.

```
    // 6 identical statements omitted here

    RectF rectImage(0.0f, 0.0f, (float)m_piISGU-
>GetItemWidthPx() - 1.0f, (float)m_piISGU->GetItemHeightPx() -
1.0f);

    // 61 identical statements omitted here
    colorBorder.SetFromCOLORREF(GetBorderColor());

    // 2 identical statements omitted here
```

```
    // 6 identical statements omitted here

    RectF   rectImage(0.0f,   0.0f,   (float)s_cxInkItem   -   1.0f,
(float)s_cyInkItem - 1.0f);

    // 61 identical statements omitted here
    colorBorder.SetFromCOLORREF(GetFrameColor());

    // 2 statements identical omitted here
```

**Figure 8. A confirmed example of code refactoring**

```
if (!pxdsi || !pxdsl)

// 13 identical statements omitted here
if (FAILED(pxdsi->HrDeleteNode(ppxslChildren[l])))

// 10 statements identical omitted here
```

```
if (!m_spxdsi || !m_spxdsl || !m_pDesc)

// 13 identical statements omitted here
if (!ParseProperty(ppxslChildren[l]))

// 10 identical statements omitted here
```

**Figure 9. A clone group tagged as "Immune"**

# 7. RELATED WORK

Research on code-clone detection has been an active research topic in recent years [3][10][17][24][27]. Roy et al. [27] conducted an extensive survey on this research topic.

In contrast to other previous approaches on code-clone detection that conduct aggressive code parameterization without imposing any constraint on characteristics of statements (e.g., CCFinder [18], CP-Miner [21], and Deckard [14]), our code-similarity metric enables users to control the degree of tolerating statement variations by parameter α, allowing XIAO to filter out many false-positive clones that other approaches would report. Our code-similarity metric also enables users to control the percentage of inserted/deleted/modified statements, allowing XIAO to detect near-miss code clones with any number of statement gaps. At the same time, the algorithm efficiency is still achieved since XIAO uses a coarse-to-fine mechanism. Token-based approaches either cannot effectively detect near-miss clones (e.g., CCFinder) or cannot efficiently detect clones with over three gaps (e.g., CP-Miner).

Clone-detection approaches based on parse tree (e.g., CloneDR [5][6] and Deckard) can detect near-miss clones with over three-statement gaps. However, in their approaches, either the percentage of shared tokens [5][6] or the feature-vector distance [14] is used to approximate the tree-edit distance. Although such approximation enables efficient detection algorithms, it leads to false positives, due to the loss of structural similarity caused by the approximation.

Our code-similarity metric also takes into account disordered statements, allowing XIAO to detect near-miss clones with disordered statements. Many other token-based detection approaches such as CCFinder or CP-Miner do not detect clones with disordered statements; parse-tree-based approaches can detect clones with disordered statements; however, they suffer from false positives.

Recently, Gabel et al. [11] proposed a scalable algorithm for detecting semantic code clones based on dependency graphs. They defined semantic code clones as isomorphic sub-graphs of the code's dependency graph. Kim et al. [19] also proposed a memory-comparison-based algorithm for code-clone detection, called MeCC. Their approach can detect near-miss code clones, including clones with disordered statements. Their focus is on detecting semantic code clones, and it is unclear how their detected code clones overlap with near-miss code clones (the focus of XIAO). Such investigation is left for future work.

Besides advances in clone detection, recent research has also made progress on applying clone detection in various software-engineering tasks such as bug detection and refactoring. Near-miss code-clone detection has been used to help identify code-refactoring opportunities [12][31] or find plagiarisms [25][26]. To search whether there are cloned copies of a piece of buggy code, Li and Ernst proposed CBCD [23], a scalable clone-search algorithm that compares graph isomorphism over program dependency graphs. At Microsoft, XIAO has also been used for searching cloned code (e.g., detection of duplicated vulnerable code) and finding refactoring opportunities; comparing to these previous approaches, XIAO is more general and can be used in broader scenarios with high tunability, scalability, compatibility, and explorability.

One important application of detecting near-miss code clones is helping engineers to identify potential bugs caused by inconsistent code changes. CP-Miner [21] detects bugs caused by inconsistently renamed identifiers. The approach by Jiang et al. [15] detects inconsistent contexts of detected code clones. Since XIAO is able to detect near-miss code clones with arbitrary gaps, XIAO has the capability of detecting more types of bugs caused by inconsistent code changes.

There are various tools for code-clone detection available as either open-source tools or commercial tools. Each one performs well in only some aspects. Most of them can detect type-I/II clones well, but have limited capability on detecting type-III clones. Few of them can detect code clones with disordered statements. Few of them provide good tunability. In contrast, XIAO can detect type-III code clones with or without disordered statements, and has high tunability on the tolerance of inserted/deleted statements.

Some of existing tools provide Graphical User Interfaces (GUI) available for exploring code clones. There exists a GUI front-end called GemX for CCFinder [18] to allow users to interactively explore clones with different metrics, such as LOC and distance of folder locations. CP-Miner provides visualization for highlighting clone differences without the concept of blocks. Simian[3] is a Similarity Analyzer for identifying duplication in code written in various languages. It provides limited explorability, displaying only one snippet from each clone group (assuming all copies from a clone group are exactly the same). CloneDR [5][6] provides a summary report and individual clone-set reports, but provides no visualization of clone differences. The uniqueness of XIAO in terms of explorability lies in supporting rich interaction and visualization: intuitive visualization of differences between cloned snippets besides allowing users to tag code clones.

There are some available tools with features of code-clone management, such as CloneTracker [9] and SimScan [4]. CloneTracker is useful for engineers to track code clones. SimScan also provides GUI for clone management and tracking, supporting simultaneous editing. XIAO's tagging mechanism can serve for similar purposes but XIAO provides both clone detection and management with high tunability, scalability, compatibility, and explorability.

The most recent related work is the work done by Jang et al. [13]. They developed a scalable approach for detecting unpatched code clones. Their approach is language agnostic and produces relatively low false-detection rate. They applied their approach on entire OS distributions. While sharing the features of high scalability and compatibility as their approach, our approach is applied on commercial codebases, and is designed to be continuously used by engineers in their daily practices. Therefore, our approach has unique features such as high tunability and explorability.

# 8. CONCLUSION

In this paper, we report our experiences of carrying out successful technology transfer of our new approach of code-clone detection, called XIAO. XIAO has been integrated into Microsoft Visual Studio 2012, to be benefiting a huge number of engineers in industry. We have discussed main success factors of XIAO: its high tunability, scalability, compatibility, and explorability. We have also presented empirical results on in-practice applying XIAO on real scenarios within Microsoft for the tasks of security-bug detection and refactoring. The results demonstrate the

[3] http://www.harukizaemon.com/simian/index.html

[4] http://blue-edge.bg/simscan/

benefits of XIAO in these tasks. In addition, it was observed that applying XIAO on detecting duplicated vulnerable code (with the target users as security engineers) has occurred much more often than the applying XIAO on refactoring (with the target users as software engineers).

## 9. ACKNOWLEDGMENTS

We thank our (former) colleagues and interns for their contribution on the implementation of XIAO: Sanhong Chen, Yan Duan, Tiantian Guo, Shi Han, Ray Huang, Qi Jiang, Feng Li, Xiujun Li, Jianli Lin, Huiye Sun, Jinbiao Xu, Jiacheng Yao, and Chiqing Zhang. We thank our colleagues for their help and joint efforts on the successful tech transfer of XIAO, especially Ian Bavey, Gong Cheng, Sadi Khan, Weipeng Liu, and Peter Provost. We thank our colleagues at Microsoft for their feedback and discussion, especially Jonus Blunck, Andrew Fomichev, Shi Han, Xiaohui Hou, Peter Nobel, Landy Wang, Jinsong Yu, and Qi Zhang. We also thank Simone Livieri for his help on evaluations of XIAO.

## 10. REFERENCES

[1] http://en.wiktionary.org/wiki/inversion_pair, as of Feb. 26, 2011.

[2] http://www.slideshare.net/icsm2011/lionel-briand-icsm-2011-keynote

[3] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. WCRE*, pages 86–95, 1995.

[4] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMSR: Program transformations for practical scalable software evolution. In *Proc. ICSE*, pages 625–634, 2004.

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

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

[7] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler: A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM 53(2):66-75, 2010.

[8] Y. Dang, S. Ge, R. Huang, and D. Zhang. Code clone detection experience at Microsoft. In *Proc. IWSC*, pages 63-64, 2011.

[9] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. ICSE*, pages 158-167, 2007.

[10] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *Proc. OOPSLA*, pages 175-190, 2010.

[11] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proc. ICSE*, pages 321-330, 2008.

[12] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. ARIES: Refactoring support tool for code clone. In *Proc. WoSQ*, pages 1-4, 2005.

[13] J. Jang, A. Agrawal, and D. Brumley. ReDeBug: Finding unpatched code clones in entire OS distributions. In *Proc. S&P*, pages 48-62, 2012.

[14] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. *Proc. ICSE*, pages 96-105, 2007.

[15] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proc. ESEC/FSE*, pages 55-64, 2007.

[16] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, Do code clones matter? In *Proc. ICSE*, pages 485-495, 2009.

[17] E. Juergens and N. Göde. Achieving accurate clone detection results. In *Proc. IWSC*, pages 1-8, 2010.

[18] 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, 2002.

[19] H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: Memory comparison-based clone detector. In *Proc. ICSE*, pages 301-310, 2011.

[20] R. Koschke. Survey of research on software clones. In *Proc. Duplication, Redundancy, and Similarity in Software*, 2006.

[21] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. *Proc. OSDI*, pages 289–302, 2004.

[22] M. Li, J. Roh, S. Hwang, and S. Kim. Instant code clone search, In *Proc. ESEC/FSE*, pages 167-176, 2010.

[23] J. Li and M. D. Ernst. CBCD: Cloned buggy code detector. In *Proc. ICSE*, pages 310-320, 2012.

[24] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *Proc. ICSE*, pages 106-115, 2007.

[25] L. Prechelt, G. Malpohl, and M. Philippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, University of Karlsruhe, Department of Informatics, 2000.

[26] R. Robbes, R. Brixtel, M. Fontaine, B. Lesner, and C. Bazin. Language-independent clone detection applied to plagiarism detection. In *Proc. SCAM*, pages 77-86, 2010.

[27] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. In Science of Computer Programming, 74(7):470-495, 2009.

[28] R. Tiarks, R. Koschke, and R. Falke. An assessment of type-3 clones as detected by state-of-the-art tools. In *Proc. SCAM*, pages 67-76, 2009.

[29] Y. Ueda, T. Kamiya, S. Kusumoto and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proc. IEEE METRICS*, pages 67-76, 2002.

[30] M. J. Wise. String similarity via greedy string tiling and running Karp-Rabin matching. Department of Computer Science, University of Sydney, ftp://ftp.cs.su.oz.au/michaelw/doc/RKR GST.ps, December 1993.

[31] L. Yu and S. Ramaswamy. Improving modularity by refactoring code clones: A feasibility study on Linux. In SIGSOFT Notes, 33(2), 2008.