MeCC: Memory Comparison-based Clone Detector

Heejung Kim†, Yungbum Jung†, Sunghun Kim§, Kwangkeun Yi†
†Seoul National University, Seoul, Korea
{hjkim,dreameye,kwang}@ropas.snu.ac.kr
§The Hong Kong University of Science and Technology, Hong Kong
hunkim@cse.ust.hk

ABSTRACT

In this paper, we propose a new semantic clone detection technique by comparing programs' abstract memory states, which are computed by a semantic-based static analyzer. Our experimental study using three large-scale open source projects shows that our technique can detect semantic clones that existing syntactic- or semantic-based clone detectors miss. Our technique can help developers identify inconsistent clone changes, find refactoring candidates, and understand software evolution related to semantic clones.

1. INTRODUCTION

Detecting code clones is useful for software development and maintenance tasks including identifying refactoring candidates [11], finding potential bugs [17, 15], and understanding software evolution [21, 6].

Most clone detectors [13, 20, 25, 9, 23] are based on textual similarity. For example, CCFinder [20] extracts and compares textual tokens from source code to determine code clones. Deckard [13] compares characteristic vectors extracted from abstract syntax trees (ASTs).

Although these detectors are good at detecting syntactic clones, they are not effective to detect semantic clones that are functionally similar but syntactically different.

A few existing approaches to detect semantic clones (e.g., those based on program dependence graphs (PDG)[23, 9, 26] or by observing program executions via random testing [14]) have limitations. PDG can be affected by syntactic changes such as replacing statements with a semantically equivalent procedure call. Hence, the PDG-based clone detectors miss some semantic clones. The clone detectability of random testing-based approaches may depend on the limited test coverage, covering only up to $60 \sim 70\%$ of software [28, 29, 36].

To detect semantic clones effectively, we propose a new clone detection technique: (1) we first use a path-sensitive semantic-based static analyzer to estimate the memory states at each procedure's exit point; (2) then we compare the memory states to determine clones. Since the abstract memory states have a collection of the memory effects (though approximated) along the execution paths within procedures, our technique can effectively detect semantic clones, and our clone detection ability is independent of syntactic similarity of clone candidates.

We implemented our technique as a clone detection tool, Memory Comparison-based Clone detector (MeCC), by extending a semantic-based static analyzer[19, 18, 12]. The extension is to support path-sensitivity and record abstract memory states. Our experiments with three large-scale open source projects, Python, Apache, and PostgreSQL (Section 4) show that MeCC can identify semantic clones that other existing methods miss.

The identified semantic clones by MeCC can be used for software development and maintenance tasks such as identifying refactoring candidates, detecting inconsistencies for locating potential bugs, and detecting software plagiarism (as discussed in Section 5.1).

This paper makes the following contributions:

- **Abstract memory-based clone detection technique:** We show that using abstract memory states that are computed by semantic-based static analysis is effective to detect semantic clones.
- **Semantic clone detector MeCC:** We implemented the proposed technique as a tool, MeCC (`http://ropas.snu.ac.kr/mecc`). We show the effectiveness of the proposed technique by experimentally evaluating MeCC.
- **Clone benchmark:** For our experimental study, we manually inspect and classify code clones of three open source projects. We make this data publicly available, and it can serve as a benchmark set for other clone related research (`http://ropas.snu.ac.kr/mecc`).

The rest of this paper is organized as follows: We first revisit and refine code clone definitions in Section 2, and then propose our approach in Section 3. Section 4 evaluates our approach, and Section 5 discusses our limitations and applications of our technique. Section 6 surveys related work, and Section 7 concludes our paper.

2. CLONE TYPES

Basically, clones are code pairs or groups that have the same or similar functionality [33, 31]. Some code clones are syntactically similar, but some are different.

Based on syntactic similarity, Roy et al. [31] classify clones into four types:

- Type 1 (Exact clones): Identical code fragments except for variations in whitespace, layout, and comments.
- Type 2 (Renamed clones): Syntactically identical fragments except for variations in identifiers, literals, and variable types in addition to Type 1's variations.
- Type 3 (Gapped clones): Copied fragments with further modifications such as changed, added, or deleted statements in addition to Type 2's variations.

- Type 4 (Semantic clones): Code fragments that perform similar functionality but are implemented by different syntactic variants.

These definitions are widely used in the literature [33, 32, 17], and we also use them in this paper.

The definitions of Type 1 and Type 2 clones are straightforward. Mostly, they are copies (from other code) that remain unchanged (Type 1) or have a small variance (Type 2). These clones can be easily detected by comparing syntactic features such as tokens in source code [20].

On the other hand, Type 4 (semantic) clones are syntactically different. Since there is no clear consensus on Type 4 clones, some researchers define subtypes of Type 4 clones such as statement reordering, control replacement, and unrelated statement insertion [33, 9, 26]. Similarly, we define subtypes of Type 4 clones as follows:

- Control replacement with semantically equivalent control structures (Refer to Figure 5.)
- Statement reordering without modifying the semantics (Refer to Figure 6.)
- Statement insertion without changing computation (Refer to Figure 9.)
- Statement modification with preserving memory behavior (Refer to Figure 7.)

Like Type 4 clones, there is no consensus on Type 3 clones. Stefan Bellon et al. [1] define Type 3 clones as all clones that are neither Type 1 nor Type 2. Similarly, in this paper, we define Type 3 clones as all clones that are not Type 1, Type 2, and Type 4 clones.

This paper proposes an abstract memory comparison-based clone detector, which can identify all four clones discussed in this section.

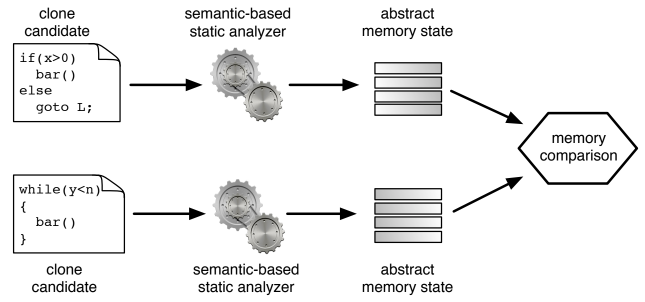
3. CLONE DETECTION BASED ON MEMORY COMPARISON

Our goal is to detect clones by comparing the functionality of code fragments regardless of their syntactic similarity. A naive way to achieve this goal is to perform exhaustive testing on a given set of clone candidates (programs). We may determine semantic similarities of programs by generating all possible inputs for programs, observing all possible executions using the inputs, and comparing their execution results. However, such exhaustive testing is often infeasible, since there might be infinite inputs and/or execution paths.

For this reason, we use semantic-based static analysis [3, 4, 37, 19, 18, 12] to determine semantic similarities of given programs, because static analysis soundly and finitely estimates the dynamic semantics of programs. In our case, we use a path-sensitive semantic-based static analyzer that symbolically estimates the memory effects of procedures.

Our overall approach is shown in Figure 1. We compute abstract memory states from given programs via static analysis. Then we compare the abstract memory states to determine code clones.

We build a semantic-based static analyzer on top of Sparrow [19, 18, 12], which can summarize each procedure after analyzing the procedure based on the abstract interpretation framework [3], and these procedural summaries have been carefully tuned to capture all memory-related behaviors in real-world C programs [19]. However, Sparrow does not support path-sensitive analysis. We extend Sparrow to be path-sensitive like [37] by adding guards and guarded values to the abstract domain.



```
if(x>0)
  bar()
else
  goto L;
```

```
while(y<n)
{
  bar()
}
```

Figure 1: Our clone detection approach: abstract memory states for each clone candidate are computed by a path-sensitive semantic-based static analyzer. These abstract memory states are compared for detecting code clones

$$\begin{array}{rcrcl}
\mathcal{M} & \in & Mem & = & Addr \xrightarrow{\texttt{fin}} GV \\
\mathcal{GV} & \in & GV & = & 2^{Guard \times Value} \\
g & \in & Guard & = & (Value \times \texttt{Rel} \times Value) \\
 & & & & + Guard \wedge Guard + \ Guard \vee Guard \\
v & \in & Value & = & \mathcal{N} + Addr + (\texttt{Uop} \times Value) \\
 & & & & +(Value \times \texttt{Bop} \times Value) + \top \\
x, \alpha, \ell & \in & Addr & = & Var + Symbol + AllocSite \\
 & & & & +(Addr \times Field) \\
 & & Var & = & Global + Param + Local
\end{array}$$

Figure 2: Abstract domains: the abstract semantics of procedure is estimated as abstract memory state over domain Mem.

The path-sensitivity is crucial for semantic code clone detection. A path-insensitive analyzer loses the relation between condition expressions and corresponding statements. For example, a path-insensitive analyzer considers the following two different `if-else` codes as the same, since it does not know which statements are belonging to which condition expressions. This insensitivity leads to detecting false positive clones.

$$\text{“}\texttt{if}(a > 0)\ A\ \texttt{else}\ B\text{”} \neq \text{“}\texttt{if}(a > 0)\ B\ \texttt{else}\ A\text{”}$$

3.1 Collecting Abstract Memory States

We compute abstract memory states at every program point of a given procedure by the conventional fixpoint iteration over abstract semantics (à la abstract interpretation [3]).

Memory State Representation Our abstract domains for memory states are presented in Figure 2. Our analysis is flow- and path-sensitive; it summarizes possible abstract memory states for each program point and all execution paths to the point. An abstract memory state ($\mathcal{M}$ in Figure 2) is a finite mapping from abstract (symbolic) addresses to guarded values. A guarded value ($\mathcal{GV}$ in Figure 2) is a set of pairs of a guard and a symbolic value, where the guard is the accumulated symbolic condition that leads to the accompanying value. The set of all variables (*Var*) consists of three disjoint sets, all global variables (*Global*), all parameters (*Param*), and all local variables (*Local*) except procedure parameters. This partitioning enables us to define three equivalence classes for variables when defining equivalent addresses in Section 3.2. Symbols (*Symbol*) are used to indicate symbolic values or symbolic addresses in global input memories of the current procedure. Allocated addresses (*AllocSite*) denote all addresses allocated at each allocation site (a static call program point for allocations). Field addresses ($Addr \times Field$) represent field variables of structures.

A symbolic value can be a number ($\mathcal{N}$), an address (*Addr*), a binary value ($Value \times \texttt{Bop} \times Value$), or a unary value ($\texttt{Uop} \times Value$). `Bop` and `Uop` denote a set of binary and unary operation symbols respectively. A guard (*Guard*) can be generated from the relations between values ($Value \times \texttt{Rel} \times Value$), where `Rel` denotes the set of comparison operators (e.g., $=$, $\leq$). Some guards can also be connected by logical operators (conjunction $\wedge$ and disjunction $\vee$).

The next step is estimating the semantics of the program as elements in this domain.

Abstract Semantics Our analysis starts from the entry point of a procedure without knowing the input memory states. The unknown input memory states are constructed by observing which locations and values are accessed by the procedure [19]. Abstract memory states are updated by evaluating each statement in the procedure, and the updates are decided by the predefined abstract semantics of each statement. For example, one abstract semantics of the assignment statement is defined as follows:

$$\frac{\mathcal{M} \vdash e_1 : \{(g,x)\} \quad \mathcal{M} \vdash e_2 : \{(g_i,v_i)\}_i}{\mathcal{M} \vdash *e_1 := e_2 : \mathcal{M}\{x \mapsto \{(g \wedge g_i, v_i)\}_i\}}$$

$\mathcal{M} \vdash e : \mathcal{GV}$ denotes that expression e evaluates to a guarded value $\mathcal{GV}$ given the memory state $\mathcal{M}$. This abstract semantics illustrates the destructive update case, in which the previous guarded values of the updated address is overwritten. The rule indicates the destructive update can happen only when the address value of e_1 is a single variable (note that the singleton set for the value in $\mathcal{M} \vdash e_1 : \{(g,x)\}$). As a result, the value of variable x is updated by the value of e_2 in memory $\mathcal{M}$. The guards for the new values are the conjunctions of guard g of the address and guards g_i of the values.

Consider the procedure `foo` in Figure 3. The abstract memory state at the exit point (line 6) is presented on the right side. At line 3, variable `res` has guarded value $\{\langle \texttt{true}, 0\rangle\}$ which means variable `res` always has the value zero at the program point. Parameter `a` is accessed in the condition expression at line 4, however the value of parameter `a` is unknown. Hence a new symbol α is created to represent the value of parameter `a`. For the field value of `a->len` which is also unknown, new symbol β is created. From the condition expression, guards $\beta > 5$ and $\beta \leq 5$ are kept for true and false branches respectively.

Inter-procedural Analysis The procedural summary information enables the analyzer to capture the semantics of procedure calls without analyzing the procedures again. At line 5, procedure `bar` is called. According to the procedural summary, the procedure returns an allocated address

```
1  int* foo(list *a,
2           int b){
3    int res = 0;
4    if (a->len > 5)
5       res = bar(b);
6    return res;
7  }
8  int* bar(int x){
9    int *m = 0;
10   if (x > 0)
11      m = malloc(x);
12   return m;
13 }
```

The abstract memory state at line 6

`a`	$\{\langle \texttt{true}, \alpha\rangle\}$
α`.len`	$\{\langle \texttt{true}, \beta\rangle\}$
`b`	$\{(\texttt{true}, \gamma)\}$
`res`	$\{\langle \beta > 5 \wedge \gamma > 0, \ell\rangle,$ $\langle \beta \leq 5 \vee (\beta > 5 \wedge \gamma \leq 0), 0\rangle$ $\}$

The procedural summary of `bar`

`x` > 0	`return alloc`
`x` ≤ 0	`return 0`

Figure 3: Procedure `bar` with its procedural summary and procedure `foo` with its abstract memory state at the exit point (line 6).

ℓ when the value of parameter `x` is greater than 0, otherwise it returns 0. The procedural summary keeps conditions (as extended from [19]) for memory behaviors of procedure. This procedural summary is instantiated with the abstract memory state at the call site (line 5). At line 5, the value of formal parameter `x` in procedure `bar` is instantiated with γ (the value of actual parameter `b`). With this instantiation of the procedural summary, we obtain the result memory state of the procedure call. Now, variable `res` points to the result guarded value, $\{\langle \beta > 5 \wedge \gamma > 0, \ell\rangle, \langle \beta > 5 \wedge \gamma \leq 0, 0\rangle\}$. Here guard $\beta > 5$ comes from the condition on true branch at line 4 and guards $\gamma > 0$ and $\gamma \leq 0$ come from the procedural summary of `bar`. At line 6, the abstract memory states on both true and false branches are joined. Variable `res` points to a guarded value $\{\langle \beta \leq 5, 0\rangle\}$ in the memory state from the false branch. The joined memory state at the return point of `foo` (line 6) is shown as the table in Figure 3. The procedural summary of procedure `foo` is automatically generated from this abstract memory state [19].

Handling Loops The termination of the fixpoint iterations is guaranteed by a widening operator [3]. Without the widening operator, fixpoint iterations may diverge because the heights of the number domain $\mathcal{N}$ and the symbolic-value domain $Value \times \texttt{Bop} \times Value$ are infinite. After five iterations (delayed widening [2]), changing values go into the special value $\top$ (indicating an unknown value). When we compare memory states, the unknown values are considered as not equivalent. Hence our clone detection may miss some clones.

Example for Comparison The abstract memory states at the exit point of procedures are compared for code clone detection. As an example, procedure `foo2` in Figure 4 is a semantic clone of procedure `foo` in Figure 3. If we disregard the names of variables, symbols, field variables, and variable types then two memories are equivalent. Note that two guards $\beta \leq 5 \vee \gamma \leq 0$ and $\beta \leq 5 \vee (\beta > 5 \wedge \gamma \leq 0)$ are equivalent. This equivalence is attained by function `simplify` [5] presented in Section 3.2.

3.2 Comparing Abstract Memory States

Given estimated abstract memory states, we need to quantify their similarities. Algorithm 1 presents the quantification steps. First, we calculate the similarities between

```
1 int* foo2(list2 *x,
2          int y){
3   int ret = 0;
4   if (x->val > 5 && y > 0)
5     ret = malloc(y);
6   return ret;
7 }
```

The abstract memory state at line 6

x	$\{\langle \texttt{true}, \alpha \rangle\}$
α.val	$\{\langle \texttt{true}, \beta \rangle\}$
y	$\{\langle \texttt{true}, \gamma \rangle\}$
ret	$\{\langle \beta > 5 \land \gamma > 0, \ell \rangle,$ $\langle \beta \leq 5 \lor \gamma \leq 0, 0 \rangle$ $\}$

Figure 4: Procedure foo2 with its abstract memory state at the exit point (line 6).

Algorithm 1: $sim_{\mathcal{M}}(\mathcal{M}_1, \mathcal{M}_2)$

```
   Input: abstract memory states $\mathcal{M}_1$ and $\mathcal{M}_2$
   Output: similarity value of $\mathcal{M}_1$ and $\mathcal{M}_2$
1  $S := \{\}$;
2  foreach address $a_1 \in dom(\mathcal{M}_1)$ do
3      foreach address $a_2 \in dom(\mathcal{M}_2)$ do
4          if $a_1 \overset{\mathcal{L}}{=} a_2$ then
               $v := sim_{\mathcal{GV}}(\mathcal{M}_1(a_1), \mathcal{M}(a_2))$;
5          else $v := 0$;
6          $S := S\{(a_1, a_2) \mapsto v\}$;
7      end
8  end
9  $best = find\_best\_matching(S)$;
10 if $\mid dom(\mathcal{M}_1) \mid + \mid dom(\mathcal{M}_2) \mid = 0$ then return 0;
11 return $\frac{2 \cdot best}{\mid dom(\mathcal{M}_1) \mid + \mid dom(\mathcal{M}_2) \mid}$
```

guarded value pairs of all possible combinations on the given memories $\mathcal{M}_1$ and $\mathcal{M}_2$ (line 2 to 8). We compare addresses using the equivalence relation $\overset{\mathcal{L}}{=}$ on addresses (as defined below). If addresses are equivalent, then we calculate the similarity of two guarded values by function $sim_{\mathcal{GV}}(\mathcal{GV}_1, \mathcal{GV}_2)$ (line 4). If addresses are not equivalent, the similarity is zero (line 5). For all combinations, the similarities of pairs are recorded in map S (line 6). Then function $find\_best\_matching(S)$ finds a subset of S that exclusively spans the two memories such that the total similarities of matched pairs becomes the biggest (line 9). Finally, the algorithm returns the ratio of similarity to the total size of memories. If both memories are empty (the denominator becomes zero), then the similarity is zero (line 10 to 11).

Equivalent Addresses Two addresses are equivalent with the relation $\overset{\mathcal{L}}{=}$ if one of the following conditions is satisfied:

$$\begin{array}{rcll} x & \overset{\mathcal{L}}{=} & y & \text{if } x, y \in Global \lor x, y \in Param \lor x, y \in Local \\ \ell & \overset{\mathcal{L}}{=} & \ell' & \text{if } \ell, \ell' \in AllocSite \\ a.f & \overset{\mathcal{L}}{=} & a'.f' & \text{if } a \overset{\mathcal{L}}{=} a' \\ \alpha & \overset{\mathcal{L}}{=} & \beta & \text{if } origin(\alpha) \overset{\mathcal{L}}{=} origin(\beta) \end{array}$$

When two variables are compared, the names and types of the variables are ignored (Var). We only check if the both variables are parameters, global variables, or non-parameter local variables. All dynamically allocated addresses ℓ are considered as equivalent regardless of their allocation sites ($AllocSite$). For field addresses ($Addr \times Field$), the names of field variables are ignored and only structural equivalences are considered. For example, $x.\texttt{val} \overset{\mathcal{L}}{=} x.\texttt{len}$ holds even if the address uses different field names. However, $(x.\texttt{next}).\texttt{len} \overset{\mathcal{L}}{=} x.\texttt{len}$ is not true because the former one has an additional field dereference. All symbolic addresses are equivalent only when their origins are the same ($Symbol$). The origin address $origin(\alpha)$ is the address pointing to symbolic address α. As an example, the following $origin(\alpha) = \texttt{a}$ and $origin(\beta) = \alpha.\texttt{len}$ hold in Figure 3.

Similarity Between Guarded Values A guarded value $\mathcal{GV}$ is a set of pairs which consist of a guard and a value. Function $sim_{\mathcal{GV}}(\mathcal{GV}_1, \mathcal{GV}_2)$ compares all guards and values in $\mathcal{GV}_1$ with those in $\mathcal{GV}_2$, and then counts the number of matched pairs n. Finally, the similarity of two guarded values are computed as follows:

$$sim_{\mathcal{GV}}(\mathcal{GV}_1, \mathcal{GV}_2) = \frac{2 \cdot n}{\mid \mathcal{GV}_1 \mid + \mid \mathcal{GV}_2 \mid}$$

$$\begin{array}{l} n = \text{maximum of } |M| \text{ s.t. } M \subseteq S \text{ and} \\ \quad \forall \langle (g_1, v_1), (g_2, v_2) \rangle \in M, (g_1, v_1) \text{ and } (g_2, v_2) \text{ appear only once} \end{array}$$

$$S = \bigcup_{(g_1, v_1) \in \mathcal{GV}_1, (g_2, v_2) \in \mathcal{GV}_2} \{\langle (g_1, v_1), (g_2, v_2) \rangle \mid g_1 \overset{\mathcal{G}}{=} g_2 \land v_1 \overset{\mathcal{V}}{=} v_2\}$$

The similarity is the ratio of the number of matched pairs to the total size of two guarded values. We seek for the maximum number of matched pairs trying to match all possible combinations ($\mid \mathcal{GV}_1 \mid \times \mid \mathcal{GV}_2 \mid$).

Equivalent Values Relation $\overset{\mathcal{V}}{=}$ establishes the equivalence on values:

$$\begin{array}{rcll} n_1 & \overset{\mathcal{V}}{=} & n_2 & \text{if } n_1 = n_2 \\ v_1 \oplus v_2 & \overset{\mathcal{V}}{=} & v_3 \oplus' v_4 & \text{if } v_1 \overset{\mathcal{V}}{=} v_3 \ \land \ (\oplus = \oplus') \land \ v_2 \overset{\mathcal{V}}{=} v_4 \\ \ominus v_1 & \overset{\mathcal{V}}{=} & \ominus' v_2 & \text{if } v_1 \overset{\mathcal{V}}{=} v_2 \ \land \ \ominus = \ominus' \\ \ell & \overset{\mathcal{V}}{=} & \ell' & \text{if } \ell \overset{\mathcal{L}}{=} \ell' \end{array}$$

Equivalence of numbers is determined by numerical equivalence ($\mathcal{N}$). Binary values are equivalent when both the pair of values and the operators are equivalent ($Value \times \texttt{Bop} \times Value$). From our definition of $\overset{\mathcal{V}}{=}$, we may miss semantically equivalent values due to their syntactic expression differences. For example, $x > 0$ and $0 < x$ should be regarded as equivalent, but it is regarded as not equivalent because of $x \overset{\mathcal{V}}{\neq} 0, > \neq <$, and $0 \overset{\mathcal{V}}{\neq} x$. To address this problem, we canonicalize the symbolic values. The canonicalization gives certain partial orders on both operators and values then sorts the binary values by the orders. Hence all semantically equivalent binary values have their unique representations.

Equivalent Guards Relation $\overset{\mathcal{G}}{=}$ determines equivalent guards:

$$\begin{array}{rcll} v_1 \sim v_2 & \overset{\mathcal{G}}{=} & v_3 \sim' v_4 & \text{if } v_1 \overset{\mathcal{V}}{=} v_3 \ \land \ (\sim = \sim') \land \ v_2 \overset{\mathcal{V}}{=} v_4 \\ g_1 & \overset{\mathcal{G}}{=} & g_2 & \text{if } \texttt{unify}(\texttt{simplify}(g_1), \texttt{simplify}(g_2)) \\ \texttt{true} & \overset{\mathcal{G}}{=} & \texttt{true} & \\ \texttt{false} & \overset{\mathcal{G}}{=} & \texttt{false} & \end{array}$$

Two relation guards $v_1 \sim v_2$ and $v_3 \sim' v_4$ in domain ($Value \times \texttt{Rel} \times Value$) are equivalent when their value pairs are the same and their relations (e.g., $<, =$) are the same. However, one formula can be presented as several different forms. For example, formulas $x > 5 \wedge (x < 10 \vee x > 0)$ and $x > 5$ look different, but are actually equivalent because $x > 5$ implies $x > 0$. To remedy this, we use a function `simplify` [5] that simplifies guards so that they do not contain any redundant sub-formulas using a decision procedure [7]. Furthermore, we want to assume $x > 5$ and $z > 5$ are equivalent if $x \overset{\mathcal{L}}{=} z$ holds. This process is done by unification algorithm `unify`, which is widely used in type systems [27]. The algorithm returns true if there exists a substitution which makes two different structures the same while preserving relations $\overset{\mathcal{L}}{=}$ and $\overset{\mathcal{V}}{=}$.

Best Matching Function $find\_best\_matching(S)$ at line 9 in Algorithm 1 finds the best matching (i.e. the matching that maximizes the sum of similarities), and then returns the maximum sum of similarities. Consider this similarity table as an example.

$\mathcal{M}_2$ \ $\mathcal{M}_1$	$(a_1^1, \mathcal{GV}_1^1)$	$(a_1^2, \mathcal{GV}_1^2)$	$(a_1^3, \mathcal{GV}_1^3)$	$(a_1^4, \mathcal{GV}_1^4)$
$(a_2^1, \mathcal{GV}_2^1)$	$\boxed{0.8}^{1}$	0.1	0.5	0.2
$(a_2^2, \mathcal{GV}_2^2)$	0.7	$\boxed{0.7}^{2}$	0.6	0.4
$(a_2^3, \mathcal{GV}_2^3)$	0.6	0.5	$\boxed{0.5}^{3}$	0.3

The boxed ones represent the best matching, since it maximizes the sum of similarities. Suppose our matching function finds this best matching. The value of *best* at line 9 in Algorithm 1 is the sum of similarities, $2 = 0.8 + 0.7 + 0.5$ of all matched pairs. Hence the similarity, $0.55 \doteqdot 2 \cdot 2/(4 + 3)$ of these two memories is returned at line 11 in Algorithm 1.

We develop a lightweight greedy algorithm to heuristically try finding the best matching which runs in $O(n^2)$, where n is the number of elements. After calculating the similarities of all pairs, the pair which has the maximum similarity is chosen as a matched one. Then the algorithm continues to choose another maximum pair among the remaining pairs until all addresses in either $\mathcal{M}_1$ or $\mathcal{M}_2$ are matched. The order of choices for the above table is annotated over the boxes. The algorithm is not guaranteed to find the best matching, but has the advantage of running in linear time. There is a combinatorial optimization algorithm called *the Hungarian method* [24] which is guaranteed to find the best matching but runs in $O(n^3)$, much slower than ours. In our experiments, we found that our algorithm yields the same results as the Hungarian method. This is because similarities of pairs are usually near 1 or 0.

3.3 Judgement of Clones

We allow parametrization by `MinEntry` to filter small clones such as a procedure containing just one line as its body. Though the similarity function $sim_{\mathcal{M}}(\mathcal{M}_1, \mathcal{M}_2)$ gives high values to similar memories, this function does not reflect the size of memories. So we give a penalty to small size memories. Note that the value of the similarity function ranges over [0, 1].

$$sim_{\mathcal{M}}(\mathcal{M}_1, \mathcal{M}_2)^{\frac{\log \texttt{MinEntry}}{\log(|\,dom(\mathcal{M}_1)\,| + |\,dom(\mathcal{M}_2)\,|)}}$$

The above formula is proportional to the size of memories and inversely proportional to `MinEntry`. Log function is used to smoothen the amount of the penalty. Here parameter `MinEntry` is given by users depending on target program size. The parameter is similar to parameter `minT` which determines the minimum number of tokens for clone candidates in Deckard [13].

We evaluate similarities for all possible pairs of abstract memories. There is a high probability that procedures with high similarity are true clones. Hence we sort all pairs according to their similarities. We allow another parameter `Similarity`, which determines the threshold of similarities of clones to be reported. If `Similarity` is set to 80% then pairs with similarity less than 0.8 are not reported.

Sometimes the similarity of two memories $\mathcal{M}_1$ and $\mathcal{M}_2$ never exceeds the given `Similarity` if there are a big difference in the entry numbers of the two memories. Hence we can skip the comparison of two memories where,

$$2 \times \frac{min(|\,dom(\mathcal{M}_1)\,|, |\,dom(\mathcal{M}_2)\,|)}{|\,dom(\mathcal{M}_1)\,| + |\,dom(\mathcal{M}_2)\,|} \leq \texttt{Similarity}.$$

This strategy significantly reduces the memory comparison time.

Users can choose parameters `MinEntry` and `Similarity` to pick thresholds to determine clones. One could set `MinEntry` high, if the one wants to ignore small clones. One could set `Similarity` high, if the one wants less false positives.

4. EXPERIMENTS

In this section, we evaluate our code clone detector MeCC. We apply MeCC to detect clones in large-scale open source projects, Python, Apache, and PostgreSQL as shown in Table 1.

Projects	**KLOC**	**Procedures**	**Application**
Python	435	7,657	interpreter
Apache	343	9,483	web server
PostgreSQL	937	10,469	database

Table 1: Properties of the subject projects.

We design our experiments to address the following research questions:

RQ1 (detectability): How many Type 3 and Type 4 clones can be detected by MeCC?

RQ2 (accuracy): How accurately (in terms of false positives and negatives) can MeCC detect clones?

RQ3 (scalability): How does MeCC scale (in terms of detection time and detectable program size)?

RQ4 (comparison): How many gapped and semantic clones identified by MeCC can be detected by previous clone detectors, CCFinder [20], Deckard [13], and a PDG-based detector [9]?

4.1 Detectability

We apply MeCC to detect clones to evaluate the detectability. In our experiments, we set `Similarity=80%` and `MinEntry =50`. Then the detected clones by MeCC are manually inspected and categorized into four clone types as discussed in Section 2 by one author who has experience with C/C++ development in industry more than eight years. The other two authors review and confirm the inspected clones.

The numbers of detected and classified clones are shown in Table 2. MeCC can detect all four types of clones. Type 4 (semantic) and some Type 3 (gapped) clones in Table 2

	Type 1	Type 2	Type 3	Type 4
Python	3	128	81	13
Apache	2	85	70	10
PostgreSQL	9	120	88	14

Table 2: The distribution of detected clone types by MeCC.

have noticeable syntactic differences. Nevertheless, MeCC can detect these clones because it only compares abstract memory states. MeCC also detects Type 1 (exact) and Type 2 (renamed) clones since syntactic similarity is usually accompanied by semantic similarity.

Figure 5 shows one Type 4 clone detected by MeCC. This is a typical example of *control replacement*. The `if-else` statements in Figure 5(a) are replaced by semantically equivalent statement using the ternary conditional '`?`:'operator in Figure 5(b). MeCC detects this clone, since their functionalities are the same and thus the abstract memory states are the same.

```
1  PyObject *PyBool_FromLong(long ok)
2  {
3    PyObject *result;
4    if (ok)
5       result = Py_True;
6    else
7       result = Py_False;
8     Py_INCREF(result);
9     return result;
10 }
```
(a)

```
1  static PyObject *
2  get_pybool(int istrue)
3  {
4    PyObject *result = istrue? Py_True : Py_False;
5     Py_INCREF(result);
6     return result;
7  }
```
(b)

Figure 5: Type 4 clone, *control replacement* from Python. The statement `if-else` is changed by using the ternary conditional `?:` operator. Syntactical differences are underlined.

A more complex Type 4 clone detected by MeCC is presented in Figure 6. The clone has two syntactic differences. One difference is *statement reordering*. Two statements from line 5 to 9 in Figure 6(a) are reordered into the statements from line 4 to 8 in Figure 6(b). The second difference comes from using intermediate variables. The local variable `sconf` is introduced at line 4 in Figure 6(a) and then used as a parameter of the `ap_get_module_config` function call at line 6. The local variable `proto` is introduced at line 10 in Figure 6(b). The return value of the `apr_pstrdup` function call at line 16 in Figure 6(b) is assigned to this variable. This value is assigned to a field address at line 18 in Figure 6 via the local variable. These syntactic changes make it difficult for textual-based clone detectors to identify such clones [20].

Understanding the semantics of procedure calls is one advantage of MeCC. An interesting Type 4 clone detected by MeCC in Figure 7 highlights this strength. The major syntactic difference between the two procedures is that the assignment statement at line 8 in Figure 7(a) is substituted by

```
1  static const char *set_access_name(cmd_parms *cmd, void *dummy,
2                                     const char *arg)
3  {
4    void *sconf = cmd->server->module_config;
5    core_server_config *conf = ap_get_module_config(
6                                     sconf, &core_module);
7
8    const char *err = ap_check_cmd_context(cmd,
9                    NOT_IN_DIR_LOC_FILE | NOT_IN_LIMIT);
10   if (err != NULL) {
11     return err;
12   }
13   conf->access_name = apr_pstrdup(cmd->pool, arg);
14   return NULL;
15 }
```
(a)

```
1  static const char *set_protocol(cmd_parms *cmd, void *dummy,
2                                  const char *arg)
3  {
4    const char *err = ap_check_cmd_context(cmd,
5                    NOT_IN_DIR_LOC_FILE | NOT_IN_LIMIT);
6
7    core_server_config *conf = ap_get_module_config(
8                    cmd->server->module_config, &core_module);
9
10   char *proto;
11
12   if (err != NULL) {
13     return err;
14   }
15
16   proto = apr_pstrdup(cmd->pool, arg);
17   ap_str_tolower(proto);
18   conf->protocol = proto;
19
20   return NULL;
21 }
```
(b)

Figure 6: Type 4 clone, *statement reordering* from Apache

the procedure `memcpy` call at line 9 Figure 7(b). Most previous clone detection techniques cannot capture this semantic similarity between a procedure call and similar assignment statements.

4.2 Accuracy

The next question is how accurately MeCC can detect clones. We manually inspected the detected clones and identified false positives, which are not real clones, but are detected as clones by MeCC.

	Total	FP	FP ratio
Python	264	39	14.7%
Apache	191	24	12.5%
PostgreSQL	278	47	16.9%

Table 3: Detected clones and false positives. Total: total number of detected clones, FP: number of false positive clones, and FP ratio: false positive ratio.

Table 3 presents the false positive clones and their ratio from three subjects (when `Similarity=80%` and `MinEntry=50`). In Python, the total number of found clones is 264, the number of false positive clones is 39, and hence the false positive ratio is around 14.7%. Similarly, the false positive ratio for Apache is 12.5%, and for PostgreSQL is around 16.9%.

```
void
appendPQExpBufferChar(PQExpBuffer str, char ch)
{
   /* Make more room if needed */
   if (!enlargePQExpBuffer(str, 1))
      return;
   /* OK, append the data */
   str->data[str->len] = ch;
   str->len++;
   str->data[str->len] = '\0';
}
```

(a)

```
void
appendBinaryPQExpBuffer(PQExpBuffer str, const char *data,
                                              size_t datalen)
{
   /* Make more room if needed */
   if(!enlargePQExpBuffer(str, datalen))
      return;
   /* OK, append the data */
   memcpy(str->data + str->len, data, datalen);
   str->len += datalen;
   str->data[str->len] = '\0';
}
```

(b)

Figure 7: Type 4 clone, *preserving memory behavior* from PostgreSQL

The most common case of false positive clones is data structure initialization. In those clones, a structure is allocated and then field variables are initialized according to the structure type. Some of them can be viewed as clones, but we scrupulously mark these initialization code pairs as false positives.

These false positive ratios look slightly higher than previous approaches [20, 13, 9]. However, one could set `Similarity` higher to reduce the false positive ratio. As an example, the false positive ratio is only 3% for Python when we set `Similarity=90%`.

In the next step, we measure the ratio of false negative clones — real clones, but missed by MeCC. For this experiment, since we need an oracle clone set, we use the benchmark provided by Roy et al. [33]. This benchmark includes three Type 1, four Type 2, five Type 3, and four Type 4 clones. We apply MeCC on the benchmark with `Similarity=80%`. Since the sizes of procedures in the benchmark are small, we set `MinEntry=2`.

	Type 1	Type 2	Type 3*	Type 4
Benchmark	3	4	5	4
MeCC	3	4	4	4

Table 4: False negatives on the benchmark set [33]. * MeCC misses only one clone.

Table 4 shows that MeCC has almost no false negatives. MeCC misses only one Type 3 clone, which has an insertion of an `if` statement that is related to a procedure call, and it changes the memory state. However, if we set `Similarity=79%`, MeCC detects this clone.

Overall, our experimental results in this section show that MeCC can detect clones accurately, with almost no false negatives and with a reasonable false positive ratio.

4.3 Scalability

In this section, we measure scalability of MeCC. We already showed that MeCC can detect clones in large-scale open source projects accurately in Section 4.1 and Section 4.2.

We measure the time spent to detect the clones for three subjects. Our experiments were conducted on an Ubuntu 64-bit machine with a 2.4 GHz Intel Core 2 Quad CPU and 8 GB RAM.

	KLOC	Analysis	Comparison
Python	435	63m32s	1m54s
Apache	343	308m58s	1m36s
PostgreSQL	937	422m04s	6m28s

Table 5: Time spent for the detection process.

Table 5 shows the results. Static analysis took about 63 minutes for Python and 422 minutes for PostgreSQL. Since our static analysis includes preprocessing, summarization/instantiation of procedural summaries, and fixpoint iterations for collecting memory states, it is computationally expensive. However, this is usually one-time cost. When software changes, we can incrementally recompute memory states of the changed parts including impacted parts according to the call relationship.

4.4 Comparison

Section 4.1 shows that MeCC can detect all four types of clones including Type 3 (gapped) and Type 4 (semantic) clones. In this section, we discuss if other clone detectors also can identify these clones.

		Python	Apache	PostgreSQL
Type 3	MeCC	81	70	88
	DECKARD	21	12	25
	CCFINDER	0	0	0
	PDG-based	10	8	11
Type 4	MeCC	13	10	14
	DECKARD	0	0	2
	CCFINDER	0	0	0
	PDG-based	1	0	1

Table 6: The numbers of detected Type 3 and Type 4 clones by MeCC, DECKARD, CCFINDER, and a PDG-based detector [9].

For the comparison, we use two publicly available syntactic clone detectors, DECKARD, a AST-based detector, and CCFINDER, a token-based detector. We also use a result set from a PDG-based semantic clone detector [9].

For DECKARD, we set the options as used in [13], `mint=30` (minimum token size), `stride=2` (size of the sliding window), and `Similarity=0.9`. For CCFINDER, we also use the default options, `Minimum Clone Length=30`, `Minimum TKS=12` (token set size), and `Shaper Level=Soft shaper`. For the PDG-based detector [9], we directly used the clone detection results provided by the authors of the detector, since the tool is not publicly available at the time of this writing.

Table 6 compares Type 3 and Type 4 clone detectability of DECKARD, CCFINDER, the PDG-based detector. We assume these detectors can detect all Type 1 and Type 2 clones, since these clones are syntactically almost the same.

CCFINDER is a scalable and fast tool which detects Type 1 and Type 2 clones accurately. However, CCFINDER could not identify any Type 3 and Type 4 clones. The main reason is that CCFinder extracts and compares syntactic tokens, but usually Type 3 and Type 4 clones are significantly different in the token level.

Deckard detects about 25% of Type 3 clones. Since Deckard uses the characteristic vectors of AST, it can detect clones with small syntactic variations. Surprisingly, Deckard identifies two Type 4 clones in PostgreSQL. The two detected Type 4 clones are classified as the *statement reordering* subtype shown in Figure 6. Since Deckard extracts characteristic vectors of these reordered ASTs, the vector only captures the number of elements in AST. However, Deckard still misses a large portion of Type 3 and Type 4 clones.

The PDG-based detector identifies about 12% of Type 3 clones. Only one Type 4 clone is identified in each Python and PostgreSQL. The detected Type 4 clones are statement reordering. Since PDG captures program semantics using data dependency and control flows, the PDG-based detector can detect some Type 4 clones like statement reordered ones.

However, these PDG-based approaches [9, 23, 26] have some limitations. (1) First, inter-procedural semantics via procedure calls cannot be supported, which means that semantic clones that differ in respect to procedure calls (e.g., function inlining) are missed. MeCC captures memory behavior of procedure calls by procedural summaries as described in Section 3. (2) Second, PDGs cannot be completely free from changes on syntactic structures, while our technique reliably determines the semantic similarity of code because we use purely semantic information (path-sensitive abstract memory effects) of programs.

Consider the semantic clone in Figure 7. We draw the PDGs of the two procedures in Figure 8 as described in the PDG-based approach [9]. The PDGs are significantly different due to following reasons: first, replacing the assignment statement at line 8 in Figure 7(a) by the procedure call at line 9 in Figure 7(b) affects the PDG, because it introduces an additional call-site node `memcpy`, which consequently introduces several child nodes. Second, adding a formal parameter `datalen` introduces new dependency flows which affect PDG.

From this observation, the PDG-based approach can miss clones made by procedure call additions or new parameter introductions, since these differences directly affect the PDG structures.

Overall, the comparison results in this section suggest that MeCC, an abstract memory-based clone detector is effective in detecting all four types (including Type 3 and Type 4) of clones.

5. DISCUSSION

We discuss potential applications and limitations of our approach. We also identify threats to validity of our experimental results.

5.1 Applications

Detecting code clones is useful for software development and maintenance tasks such as identifying or refactoring redundant code fragments [16, 11], detecting inconsistencies [17, 15], and understanding software evolution [21, 6]. MeCC allows developers to perform these tasks on semantic clones, which would be missed by previous approaches [20, 13, 9].

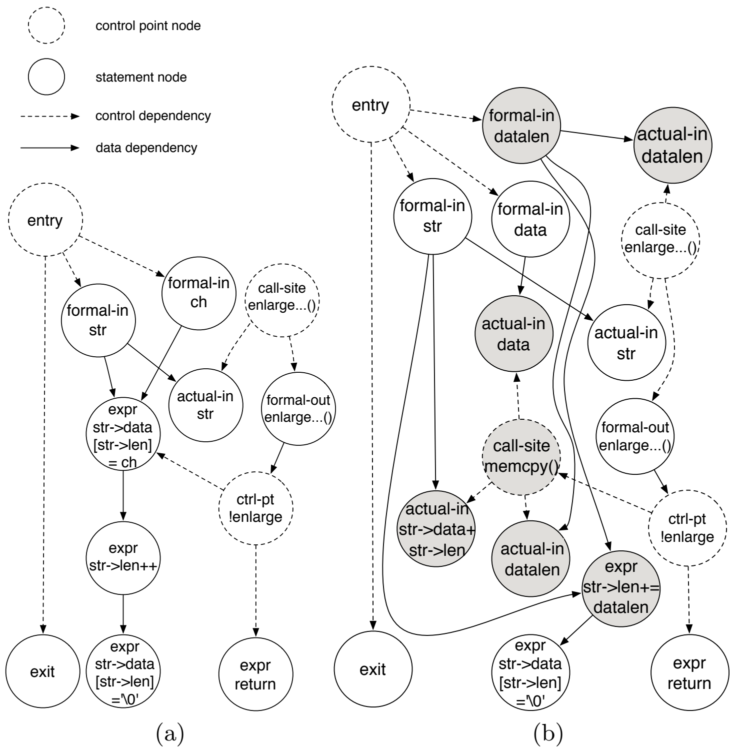


Figure 8: Two PDGs of semantic clones in Figure 7. The graphs look significantly different even though two clones are semantically similar. Grey-colored nodes are newly introduced due to changes between the two procedures.

Usually clones are results of code copies and their changes. Inconsistencies of copied code may lead to Type 3 and Type 4 clones. Often inconsistencies cause potential bugs or code smells[10].

We investigate feasibility of using MeCC to identify potential bugs and code smells caused by inconsistencies. Figure 9 shows one Type 4 clone identified by MeCC. This clone shows an inconsistency: the procedure `PQparameterStatus` in Figure 9 (b) checks whether the second parameter `paramName` is not null but the procedure `GetVariable` in (a) does not check. This inconsistency is an indicator of the existence of potential bugs, which are exploitable by passing null for the second parameter, `name`.

	# Type 3 & Type 4	Exploitable bugs (%)	Code smells (%)
Python	95	26 (27.4%)	23 (24.2%)
Apache	81	8 (9.9%)	27 (33.3%)
PostgreSQL	102	21 (20.6%)	20 (19.6%)
Total	278	55 (19.8%)	70 (25.2%)

Table 7: Exploitable bugs and code smells in Type 3 and Type 4 clones found by MeCC.

We manually inspect all Type 3 and Type 4 clones identified by MeCC to check if they are caused by inconsistencies, and if these inconsistencies lead to potential bugs or code smells. When we identify problems caused by inconsistencies, we classify them in two categories: Exploitable bug:

```
const char *
GetVariable(VariableSpace space, const char *name)
{
  struct _variable *current;
  if (!space)
     return NULL;
  for (current = space->next; current; current = current->next)
  {
    if (strcmp(current->name, name) == 0)
    {
      return current->value;
    }
  }
  return NULL;
}
```

(a)

```
const char *
PQparameterStatus(const PGconn *conn, const char *paramName)
{
  const pgParameterStatus *pstatus;
  if(!conn || !paramName)
     return NULL;
  for (pstatus = conn->pstatus; pstatus != NULL; pstatus = pstatus->next)
  {
     if (strcmp(pstatus->name, paramName) == 0)
        return pstatus->value;
  }
  return NULL;
}
```

(b)

Figure 9: Type 4 clone, *statement insertion without changing computation* from PostgreSQL, which has a potential bug due to an inconsistency.

A bug or crash is exploitable, for example, by invoking the method, or by passing certain variables for its arguments. Code smell: There are no directly exploitable bugs, but refactoring or consistent changes (with other clone pairs) are highly recommended.

Table 7 shows manual inspection results[1]. Among 278 Type 3 and 4 clones, 55 exploitable bugs and 70 code smells are found. About 45% of Type 3 and Type 4 clones are either exploitable bugs or code smells. Note that these bugs and code smells would be missed by previous approaches [20, 13, 9], since most of these Type 3 and type 4 clones were not detected by them as discussed in Section 4.4.

Overall, Table 7 implies that MeCC and its identified Type 3 and Type 4 clones are very useful to detect inconsistencies, exploitable bugs, and code smells.

MeCC can be used for plagiarism detection and common bug pattern identification. Syntactic plagiarism detection tools (e.g. Moss [35] and JPlag [30]) cannot detect plagiarism if code is copied and intentionally changed with some syntactic obfuscations. MeCC is able to detect plagiarism as long as the semantics of the copied code remains similar regardless of its syntactic changes. Similarly, MeCC can help identify common bug patterns. Kim et al. proposed BugMem [22], which identifies common bug fix patterns and locates similar bugs in other code. However, they only capture syntactic bug patterns using tokens of code. MeCC can improve their work by identifying common semantic bug patterns.

5.2 Limitations

Since, our current implementation compares abstract memory states at the exit points of procedures, MeCC detects only procedure-level clones. However it is possible to extend MeCC to find clones with a finer granularity such as basic blocks adapting a code fragments generation technique [14] to prepare code clone candidates of finer granularity. Then we can calculate every abstract memory state for each candidate and compare them to identify clones.

Collecting abstract memory states from programs is a computationally expensive task in both time and memory. Analyzing the semantics of programs takes longer than syntactic comparison. However, the current implementation of MeCC showed that MeCC scales to detect clones in PostgreSQL, which is around 1M LOC.

Similar abstract memory states do not always imply similar concrete behaviors, which may cause false positives. In the abstract interpretation framework [3], one element in an abstract domain can represent several concrete elements. Procedural summaries record memory related behaviors [19], but do not capture all concrete procedure behaviors. This limitation is inevitable since determining semantic equivalence between two programs is generally undecidable [8].

5.3 Threats to Validity

We identify the following threats to validity to our work:

Projects are open source and may not be representative. The three projects used in this paper are all open source and not representative of all software systems, and hence we cannot currently generalize the results of our study across all projects. However, these projects are chosen because they are commonly used in other code clone related research.

Manually inspected and classified clones. One author manually inspected and classified clones, and they are used to evaluate MeCC. Since there is no consensus about Type 3 and Type 4 clones, there is ambiguity in the classified clones. However, two other authors confirmed the classified clones, and we made this data publicly available.

Default options are used. DECKARD, CCFINDER, and the PDG-based detector have various options to tune their clone detectability. In this paper, we use their default options. However, careful option tuning may allow these tools to detect more Type 3 or Type 4 clones.

6. RELATED WORK

Most clone detection techniques are syntactic clone detectors ones [33, 31, 13, 20, 25, 34, 9, 32, 1] leveraging line-based [34], token-based [20, 25], or tree-based [13] approaches. These detectors are good at identifying Type 1 and Type 2 clones, but they miss most of the Type 4 and some of the Type 3 clones as discussed in Section 4.4.

Existing semantic clone detectors have limitations. For example, as we discussed in Section 4.4, PDG-based detectors [9, 23, 26] miss some semantic clones due to, for example, ignorance of inter-procedural semantics. A PDG-based technique [9] maps slices of PDGs to syntax subtrees and applies DECKARD [13] to detect similar subtrees. Although slicing enables one to detect more gapped clones, clones in each clone cluster still need to be syntactically similar. Jiang

[1]These results are also available at `http://ropas.snu.ac.kr/mecc`

et al. [14] proposed a clone detector using random testing techniques. They conclude two code fragments are clones when their outputs are the same just for a number of randomly generated inputs. Since random testing cannot cover all program paths or inputs - usually around up to 60 ∼ 70% [28, 29, 36], false positives are inevitable. Furthermore, the inter-procedural behaviors are not considered in their approach.

7. CONCLUSIONS AND FUTURE WORK

We proposed an abstract memory-based code clone detection technique, presented its implementation, MeCC, and discussed its applications. Since MeCC compares abstract semantics (as embodied in abstract memory states), its clone detection ability is independent of syntactic similarity. Our empirical study shows that MeCC can accurately detect all four types of code clones. We also show that most of Type 4 and some of Type 3 clones identified by MeCC cannot be detected by previous approaches [20, 13, 9].

We anticipate that MeCC will allow developers to find inconsistencies as shown in Section 5.1, identify refactoring candidates, and understand software evolution related to semantic clones which would be neglected by previous approaches.

Even so, we still see room for improvement. Since MeCC uses static analysis, it requires some time to analyze the entire source code prior to our clone detection process. Our static analyzer can only collect memory states in the procedure level, and thus MeCC can detect only procedure level clones. To detect block level clones, we plan to adapt our static analyzer to collect memory states for any arbitrary code blocks.

Overall, we expect that future clone detection approaches will exploit more deep semantics of code via static analysis program logic, and/or other program verification technologies. MeCC is one step forward in this direction.

8. ACKNOWLEDGMENTS

Our thanks to Lingxiao Jiang, Mark Gabel, and Zhendong Su for providing DECKARD, PDG-based code clone results, and valuable feedback on our research.

9. REFERENCES

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI '03*, pages 196–207, 2003.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252, January 1977.

[4] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI'08*, pages 270–280, June 2008.

[5] I. Dillig, T. Dillig, and A. Aiken. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In *SAS '10*, Lecture Notes in Computer Science, 2010.

[6] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE '07*, pages 158–167, 2007.

[7] B. Dutertre and L. D. Moura. A fast linear-arithmetic solver for dpll(t. In *CAV '06*, pages 81–94. Springer, 2006.

[8] P. E. G. Cousineau. Program equivalence and provability. In *MFCS '79*, 1979.

[9] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE '08*, pages 321–330, 2008.

[10] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *OOPSLA '10 (to appear)*, 2010.

[11] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, and K. Words. Aries: Refactoring support environment based on code clone analysis. In *SEA '04*, pages 222–229, 2004.

[12] Y. Jhee, M. Jin, Y. Jung, D. Kim, S. Kong, H. Lee, H. Oh, and K. Yi. Abstract interpretation + impure catalysts: Our sparrow experience. In *Workshop of the 30 Years of Abstract Interpretation*, 2008.

[13] L. Jiang, G. Misherghi, and Z. Su. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07*, pages 96–105, 2007.

[14] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA '09*, pages 81–92, 2009.

[15] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE '07*, pages 55–64, 2007.

[16] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON '93*, pages 171–183, 1993.

[17] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE '09*, pages 485–495, 2009.

[18] Y. Jung, J. Kim, J. Shin, and K. Yi. Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis. In *SAS '05*, volume 3672 of *Lecture Notes in Computer Science*, pages 203–217, 2005.

[19] Y. Jung and K. Yi. Practical memory leak detector based on parameterized procedural summaries. In *ISMM '08*, pages 131–140, 2008.

[20] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28:654–670, 2002.

[21] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, 2005.

[22] S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of bug fixes. In *SIGSOFT '06/FSE-14*, pages 35–45, 2006.

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

[24] H. W. Kuhn. The hungarian method for the assignment problem. In *50 Years of Integer Programming 1958-2008*. Springer Berlin Heidelberg, 2009.

[25] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[26] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *KDD '06*, pages 872–881, 2006.

[27] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[28] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .net with feedback-directed random testing. In *ISSTA '08*, pages 87–96, 2008.

[29] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE '07*, pages 75–84, 2007.

[30] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, 8:1016–1038, 2001.

[31] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY*, 115, 2007.

[32] 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.

[33] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[34] B. S.Baker. A program for identifying duplicated code. In *Computer Science and Statistics: Proc. Symp. on the Interface*, pages 49–57, March 1992.

[35] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *SIGMOD '03*, pages 76–85, 2003.

[36] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Mseqgen: object-oriented unit-test generation via mining source code. In *ESEC/FSE '09*, pages 193–202, 2009.

[37] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *ESEC/FSE-13*, pages 115–125, 2005.