Searching for Better Configurations: A Rigorous Approach to Clone Evaluation

Tiantian Wang
Harbin Institute of Technology
Harbin, China

Mark Harman, Yue Jia and Jens Krinke
CREST, University College London
London, UK

ABSTRACT

Clone detection finds application in many software engineering activities such as comprehension and refactoring. However, the confounding configuration choice problem poses a widely-acknowledged threat to the validity of previous empirical analyses. We introduce desktop and parallelised cloud-deployed versions of a search based solution that finds suitable configurations for empirical studies. We evaluate our approach on 6 widely used clone detection tools applied to the Bellon suite of 8 subject systems. Our evaluation reports the results of 9.3 million total executions of a clone tool; the largest study yet reported. Our approach finds significantly better configurations ($p < 0.05$) than those currently used, providing evidence that our approach can ameliorate the confounding configuration choice problem.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Algorithms, Experimentation, Measurement

Keywords

Clone Detection, Genetic Algorithm, SBSE

1. INTRODUCTION

Software contains clones. Several studies [18, 32, 22] have provided evidence for widespread presence of clones and there has been much interest and previous work on software engineering applications and implications such as software evolution [21, 36, 7, 3, 29], refactoring [4], and bug detection [24, 31, 23]. Many tools and techniques for detecting clones have been studied [34]. They use a variety of approaches, based on raw text, lexical tokens, syntax trees, metrics, or graph-based structures – sometimes combined [8]. The wide variety of approaches has led to many comparative studies of detection techniques [2, 34, 29]. It was widely believed that cloned code is a code smell and many empirical studies of clone properties [20, 21, 26, 36, 15] have investigated differences between cloned and non-cloned code.

Unfortunately, all these previous studies suffer from an effect we term the 'confounding configuration choice problem'. That is, the validity of results on the properties of clones are threatened by the confounding effect of the configuration choice: How do we know that differences observed are related to the properties of clones, rather than the properties of configuration choices? Similarly, differences observed in comparative studies may merely be a manifestation of the configuration choice adopted in the study, for example, by using different configurations for the minimal clone size.

We studied the confounding configuration choice problem by surveying the clone detection literature and report the results in this paper as a motivation for our work. Our survey revealed that, of 274 software clone papers (from the clone literature repository [35]), 185 include an empirical experiment, of which 113 (61%) explicitly comment on problems arising from the effects of the confounding configuration choice problem. We suspect that others may be affected but have not commented explicitly on it in their work.

In order to ameliorate the confounding configuration choice problem we introduce a search based approach to find the configurations for clone detection techniques that yield maximal agreement. This is the first time that Search Based Software Engineering (SBSE) has been used as a way to augment the robustness of an empirical study of software engineering. Recent work on SBSE has considered other forms of parameter tuning for data mining [30] and traceability [25]. Whereas these two recent contributions seek to help the tools' end users, we seek to aid empirical software engineering researchers' evaluation of the approaches the tools implement. Our goal is to provide an additional approach to experimental rigour. We believe that this "search for suitable parameter configurations" will find other applications in empirical software engineering.

We implemented our approach as a desktop application EvaClone. We also implemented a cloud-based parallelised version called CloudEvaClone. We used CloudEvaClone to conduct the large-scale empirical study needed to evaluate our approach; the largest hitherto reported in the literature.

The paper makes the following four primary contributions:
Justification and Motivation: A detailed review of the literature that exposes the concern repeatedly expressed by previous experimenters regarding the threats arising from the confounding configuration choice problem.

ESEC/FSE '13, August 18–26, 2013, Saint Petersburg, Russia

http://doi.acm.org/10.1145/2491411.2491420

EvaClone and CloudEvaClone: Desktop and cloud-based implementations of our search based solution to the confounding configuration choice problem on which we report and which we make available to the research community.
Empirical Evaluation: An empirical study, the results of which show that (1) Default configurations perform poorly as a choice of configuration for clone experiments. (2) Our approach produces significantly better configurations, thereby reducing the confounding configuration choice problem.
Optimised Configurations: The configurations found in the empirical evaluation can be used for future agreement-based clone evaluations without running EvaClone.
Actionable Findings: Failure to consider confounding configurations compromises the scientific findings of empirical studies of clone detection. Future studies of clones can use (Cloud)EvaClone to find appropriate configurations.

The rest of the paper is organised as follows: Section 2 presents a focussed review of empirical clone studies, demonstrating the importance of the confounding configuration problem and the need to cater for confounding effects in clone tool 'agreement studies'. Section 3 introduces the EvaClone tool and its cloud-virtualised sister CloudEvaClone. Section 4 describes the experimental methodology we adopt for our empirical study of CloudEvaClone applied to the clone detection agreement problem. Section 5 presents and discusses the empirical study and Section 6 concludes.

2. BACKGROUND AND MOTIVATION

The motivation for our work is that there is a problem at the very heart of the way in which clone evaluation has taken place in the literature; the configurations adopted are often arbitrary or unspecified, yet configuration choices have a significant impact on the behaviour of the tools compared. The scientific justification for our claim rests on two kinds of evidence: a detailed review of the literature and the presentation of empirical evidence that configuration choices can lead to significantly different results. In this section we summarise the results of the literature review. The remainder of the paper is concerned with the evidence that the configurations have a significant impact on behaviour and the introduction of our approach to overcome the difficulties that this imposes on any attempt at rigorous empirical evaluation.

We reviewed the 274 papers on clone analysis, management, and detection, available at the time of writing in the widely-used clone literature repository [35]. Among these 274 papers, we found that 89 papers have no empirical study of clone detection behaviour, so our analysis focused on the remaining 185 papers, all of which include an empirical study concerning at least one code clone detection tool. Among these 185 papers, 113 (61%) report that the experimenters are aware that tool configuration (parameter selection and threshold settings) may have affected the results reported in the paper. In particular, among the 57 papers that contain a specific section discussing 'threats to validity', 75% of these (43 papers) consider this a problem that needs to be taken into account. This provides evidence that the research community recognises the problem that we address in this paper.

It is widely believed that the results of any clone experiment will not only depend on the choice of tools, studied systems, or analysed language, but will also depend on the configuration of the tools used [27, 28], leading many authors to attempt to cater for variations and reporting results for multiple clone detection tools.

Table 1: Categories of Multiple Clone Tool Studies

Category	Cases	Papers	Total
agreement / disagreement	Use the other tools to evaluate the given clone detection tool	31 (58.5%)	81.1%
	Compare how results are different from detection tools	10 (18.9%)	
	Select the best tool for the analysis task	2 (3.8%)	
other cases	Use the union of the clone detection results	5 (9.4%)	18.9%
	Only compare the execution time between different tools	5 (9.4%)	

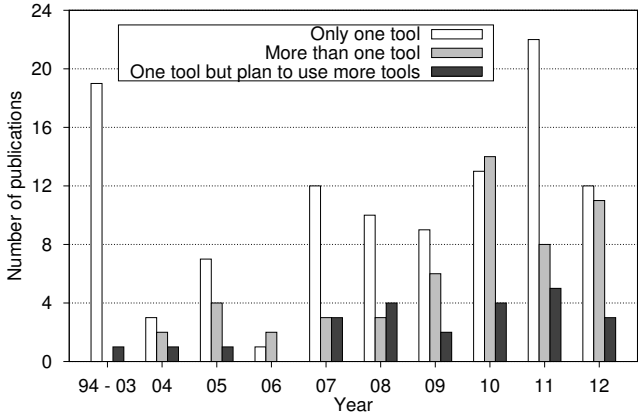

Figure 1: Number of Tools Used in Previous Work

As can be seen from the analysis of the literature in Figure 1, after 2003, more and more papers have paid attention to multiple clone detection tools. In total, 53 papers have used multiple tools, while a further 24 papers used a single tool but reported that the authors planned to use multiple tools in their future work to improve result validity.

Summary details concerning the 53 papers that used multiple clone tools in their experiments are presented in Table 1. As the table reveals, 81% of these papers (43 papers in total) are concerned with the analysis of the agreement (or disagreement) among the results produced by the different clone detection tools studied. For all of the 43 papers, the configuration of the tools will be of paramount importance.

Table 2 shows the choices of approach to configuration adopted by each of the 43 papers. There are two cases: either the paper compares the authors' *own* tool against *other* tools or the paper is an empirical study using a set of *other* tools (with no 'own' tool).

Our survey revealed six distinct styles of approach to handle the confounding configuration choice problem in these 43 studies. Table 2 lists these six approaches in increasing order of rigorousness, according to the degree to which the confounding effect is accounted for in the experiments. Therefore, those entries to the bottom right of Table 2 can be regarded as the 'most rigorous'.

Table 2: Approaches to the Confounding Configuration Choice Problem in 43 Clone Comparison Papers

		Other Tools					Total
		Und	*Arb*	*Def*	*Jus*	*Var*	
Own Tools	*N/A*	2	0	4	3	1	10
	Und	2	0	0	0	0	2
	Arb	2	0	0	0	0	2
	Def	0	0	1	0	0	1
	Jus	1	0	5	8	1	15
	Var	2	0	5	4	2	13
Total		9	0	15	15	4	43

The entries in Table 2 have the following meanings:

1. *N/A*: no own tool is used.
2. *Und*efined: configurations are not reported.
3. *Arb*itrary: configurations are reported but with neither justification nor explanation.
4. *Def*ault: the tools' default configurations are used.
5. *Jus*tified: configurations are reported, together with some explanation as to why they have been selected.
6. *Var*ied: several different configurations are used to attempt to cater for confounding configuration effects.

As Table 2 reveals, many authors simply use defaults, while very few attempt to experimentally cater for the potential confounding effects of parameter choice. The clone detection community does recognise this issue. Indeed, it is frequently commented upon in the literature. For example Nguyen et al. [29], who evaluated the performance of their tool JSync, comparing it to Deckard [16] and CCFinder [18] warn "*Even though we ran JSync with different thresholds in comparing, we used the default running parameters for CCFinder and Deckard, thus, their results might not be as good as those with tuning for each system*" [29]. Kim et al. [19] also compared their tool, MeCC, to Deckard, CCFinder, as well as the PDG-based detector [9] adding the caveat "*We use their default options. However, careful option tuning may allow these tools to detect more Type-3 or Type-4 clones*" [19].

Not all papers are concerned with the introduction of a new clone detection tool. Several empirically evaluate the effects that clones have on software engineering activities. However, all such empirical studies are affected in precisely the same way as the tool evaluation studies. For example, Hotta et al. [15] investigated the effects of clones on software evolution, but were compelled to include a similar warning about the confounding effects of configuration choices on their findings, warning the reader that "*In this empirical study, we used default settings for all the detection tools. If we change the settings, different results will be shown*" [15].

The clone research community is not complacent about the confounding configuration choice problem, it is merely that no approach has been found to ameliorate its effects and thereby to provide a more rigorous approach to empirical clone studies. This is the problem addressed by this paper.

Ideally, any clone-related study would compare against a gold standard, i.e. an oracle that knows whether code is cloned. However, such an oracle is not available. There have been attempts to create benchmarks, but even the large-scale Bellon study [2] has only created a small sample of confirmed clones. Moreover, even experts can often not agree on whether something is a clone. As a result, authors of empirical studies of cloned code use clone detectors themselves as oracles and accept their inaccuracy. Using multiple clone detectors in such studies improves on this: It is very likely that something is actually cloned if all used tools agree that it is a clone and it is very likely that something is actually not cloned if all tools agree that it is not cloned.

We introduce a search based approach to determine the configurations (from the space of all configuration choices) that maximises tool agreement for a set of tools on a set of systems. Such an 'agreement optimised' configuration avoids the problem that different tools might disagree about whether or not a line of code is cloned, simply because of the configuration parameter choices. Where tools disagree on the agreement optimised configuration, the experimenter can have a stronger confidence that this is due to the effect studied and not some coincidence of the parameter settings. We believe that this work will put clone evaluation on a more rigorous and firm footing.

3. THE EVACLONE SOLUTION

In this section we introduce EvaClone, our approach to addressing the confounding configuration choice problem and its implementation. We then briefly describe the sister version which is parallelised for cloud deployment.

3.1 EvaClone

Given a clone detection tool set $TS = \{T_1, \ldots, T_n\}$, a set of subject systems $SS = \{S_1, \ldots, S_m\}$, the clone detection tool configuration problem is to automatically search for configuration settings, X, for TS in the configuration search space Ω, subject to:

$$\left.\begin{array}{ll} maximise & f(TS(X), SS) \\ & X \in \Omega \end{array}\right\}$$

The fitness function f can be defined according to the specific clone analysis task. In this paper we report on the application of EvaClone using fitness functions that seek to maximise tool agreement, since this is important for many of the empirical studies in the literature, as revealed by the review in Section 2. However, the choice of fitness function is a parameter to our approach and so EvaClone provides a framework for optimising configurations of clone detection tools.

Our tool uses a Genetic Algorithm (GA) to search the configuration space, guided by the chosen fitness function. Figure 2 depicts the EvaClone architecture. Phase 1 (Initialisation) generates a randomised initial population of configurations. This population is seeded with defaults to give the existing default a fair chance of selection.

The computational expense of the approach resides in Phase 2 (Fitness Evaluation), since this requires that each tool is executed with each configuration from the population on each subject system. The fitness function gives a value to each configuration by evaluating the level of agreement among the tools for that specific configuration. This phase includes the conversion of the clone detector output into a General Common Format (GCF); any output in a proposed standard format (RCF) [11] can be converted to GCF.

The genetic operations used in Phase 3 are described in Section 4.2. The termination condition is satisfied when no fitness improvement occurs for thirty generations or when the maximum allowed budget (100 generations in our study) is exhausted. The final recommended configuration is the one found with the highest fitness over all generations.

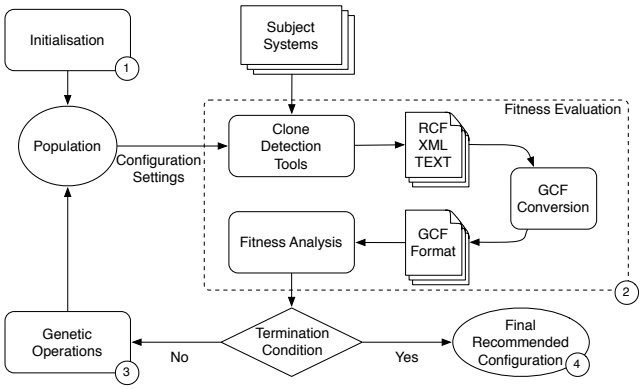

Figure 2: EvaClone Architecture

3.2 A Parallelised Cloud-Deployed EvaClone

EvaClone needs to run multiple clone experiments on multiple tools over multiple subject systems to find suitable configurations. Depending on the subject systems' size and the performance of the clone detection tools, it can take from 30 seconds to 4 minutes to evaluate one individual in one experiment. Each experiment uses a population size of 100 and runs up to 100 generations.

Furthermore, in order to evaluate our approach, we need to perform inferential statistical testing on the results required for evaluation of EvaClone configurations [1, 13]. This statistical testing requires multiple experimental trials for each experiment. In this paper we used 20 such repeated trials for each experiment on which we report. On a conventional desktop machine, this would have taken a total time of approximately 15 years, which is clearly infeasible.

In order to cope with the computational time required, we developed a parallel version of EvaClone which can be deployed in the cloud, thereby allowing flexible control of computational time. Our approach also allows the EvaClone user to trade computational cost for execution time. We call this parallel virtualised version 'CloudEvaClone'. Using CloudEvaClone's parallel computation, spread over many virtualised machine instances, we were able to reduce computation time for all experiments reported on in this paper from a minimum possible time of 15 years (on a single desktop) to 2 weeks (in the cloud).

The virtualisation required to implement CloudEvaClone also facilitates *perfect facsimile* replication of our results. That is, we are able to make available the entire virtualised execution image for our experiments to other researchers. Our image can be re-deployed and re-executed by others, thereby avoiding the confounding effects (due to different machines, operating systems and platforms) that bedevil empirical replication attempts for software tool studies. Using our approach, authors of future studies can thus have greater confidence that they are comparing 'like with like'.

CloudEvaClone consists of two tiers, a server tier and a client tier, the architecture of which is depicted in Figure 3. The server tier maintains a database and a task list. The database stores the previous evaluated fitness values, thereby memoising previous results to avoid the cost of re-computation. The task list maintains a list of current evaluation tasks, which is used to distribute the evaluation cost in parallel. The client tier runs two kinds of instances: master instances and slave instances. Each master instance runs one experiment with a parallel GA algorithm.

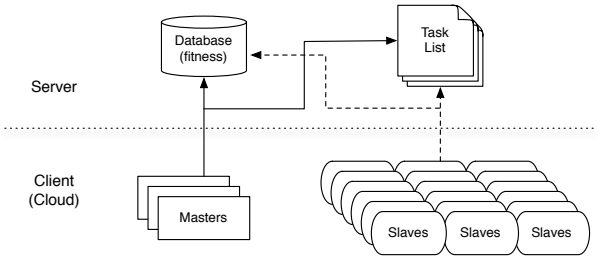

Figure 3: CloudEvaClone Architecture

For each generation, the algorithm first searches the database for the fitness of the current individual. If no record is found, the algorithm creates an evaluation task and sends it to the task list. Slave instances repeatedly poll the task list for unassigned tasks. If a slave finds tasks in the list, the first task is popped from the list and the slave will evaluate the task and send the fitness value back to the database in the server tier.

Since we are searching the space of all possible configurations of each tool, we are also, implicitly, testing the clone detection tools (very thoroughly in fact). As a result of this rigorous testing, it can happen that CloudEvaClone finds configurations for which a tool either crashes or non-terminates (these tools are research prototypes and not robust industrial strength tools after all). To avoid non-termination of a clone detection tool resulting in overall non-termination of the whole approach, both EvaClone and CloudEvaClone allow the user to set a conservative threshold execution time after which execution is automatically aborted. We performed an initial sensitisation study which suggested that when a tool runs for more than 10 minutes on a clone detection instance, it tends to run for at least several further hours without result, so we set our cut-off threshold at 10 minutes.

CloudEvaClone is designed to be flexible; the master and slaves do not interact with each other directly. This design means that, subject to the trade off between the user's time and budget, master and slave instances can be added and/or removed from a CloudEvaClone execution at any time without the need to restart the experiment.

For the empirical study reported in this paper, we deployed CloudEvaClone on Amazon's Elastic Compute Cloud (EC2). Three master instances were used, running three experiments at a time and two hundred slave instances were used to perform fitness evaluations. All instances were run on Ubuntu 12.04 LTS systems. The specifications of each instance are: 4 EC2 Compute Units with 7.5 GB memory and 850 GB instance storage.

In total one or other clone detector was executed a total of 9.3 million times. Memoisation was deployed in 13% of cases, thereby saving this proportion of the execution time. Since the clone configuration search space is approximately 10^{17} we cannot expect higher figures for memoisation, but the saving it represents is valuable, since it directly corresponds to reductions in cloud computation costs. Though it is executed in the cloud, CloudEvaClone is merely a parallelisation of EvaClone; it produces the same results and has the same functional properties (including memoisation). CloudEvaClone's master/slave parallelisation simply allows the non-functional properties of cost and execution time to be traded off against one another in a fully flexible manner.

Table 3: The Bellon Suite of Eight Benchmark Subjects for Clone Detection Research Used in this Paper

Subjects Sets	Subjects	Full Name	Files	LOC	SLOC	Description
C Set	*weltab*	weltab	65	11,700	10,038	election tabulation system
	cook	cook	590	80,408	50,621	file construction tool
	snns	snns	625	120,764	86,539	simulator for neural networks
	psql	postgresql	612	234,971	156,312	Database management system
Java Set	*javadoc*	netbeans-javadoc	101	14,360	9,579	Javadoc package in the NetBean IDE
	ant	eclipse-ant	178	34,744	16,106	Ant package for the Eclipse IDE
	jdtcore	eclipse-jdtcore	741	147,634	98,169	JDT core package for the Eclipse IDE
	swing	j2sdk1.4.0-javax-swing	538	204,037	102,836	Java 2 SDK 1.4.0 Swing components

Table 4: The Clone Detection Tools Used

Tool	Approach	Support Language	Type
PMD's CPD 5.0 [6]	Token	C, C++, C#, Java	1,2
IClones 0.1 [10]	Token	Java, C, C++, ADA	1,2,3
CCFinder 10.2.7.4 [18]	Token	C, C++, Java, COBOL	1,2,3
ConQAT 2011.9 [17]	Token	independent	1,2
Simian 1.5.0.13 [14]	Text	independent	1,2
NiCAD 3.2 [33]	Parser	C, C#, Java, Python	1,2,3

4. EMPIRICAL EVALUATION

The choice of fitness function, clone tools to be configured, and subject systems to which these tools will be applied are all parameters to our approach. However, we wish to provide a concrete evaluation. Therefore, we chose a fitness function, detection tool set and subject system set to address the problems of rigorous empirical evaluation of clones in experiments for which clone detection agreement matters. As shown in Section 2, such configurations have proved to be pivotal in removing the confounding configuration problem for at least 43 previous empirical studies. Therefore, such 'maximally agreeing' configurations are also likely to remain important in future empirical clone evaluation work.

4.1 Clone Detectors and Subject Systems Used

For our evaluation we used six clone detectors, all of which are commonly used in clone analysis work. Table 4 presents summary information about each of these tools. For the systems on which to apply the tools, we adopted the widely-used Bellon benchmark [2]. These choices were made to maximise the chances that the specific configuration results reported in this paper will be a useful contribution to the community in future work on clone evaluation.

Summary information about the subject systems used is presented in Table 3, with data about the systems' sizes (in lines of code, physical source lines, and number of files). The subject systems are divided into two sets: a set of subject systems written in C and a set of systems written in Java.

4.2 Configuration of the Genetic Algorithm

Clone detection researchers have a consensus that clone fragments containing fewer than 6 lines are meaningless [2]. Search based tools (such as EvaClone and CloudEvaClone) are well-known to exploit search spaces to find solutions that maximise fitness irrespective of such domain-specific concerns, unless they are factored into the search problem [12]. Therefore, we constrained the search to a range of minimum lines (*MinLine*) of a clone fragment from 5 to 7. For ConQAT, Simian, and NiCAD, the minimum length of clones is measured in lines, so clones no smaller than *MinLine* are output by these tools. However, for PMD, IClones and CCFinder, the minimum length is measured in tokens. There is no simple correspondence between tokens and lines: A line may contain few or many tokens. To solve this compatibility problem between tools, we set the range of minimum tokens (*MinToken*) to 10–300. The lower bound of *MinToken* is sufficiently small that it ensures that clones smaller than *MinLine* are possible. Should any clone reported by any tool contain fewer than *MinLine* lines, we remove it using a post-processing filter applied to all tools. Thus, the *MinLine* setting is enforced over all tools.

Each configuration setting is coded as an integer in the range specified in Table 5 (which also shows the final configurations reported by CloudEvaClone, which are discussed in Section 5). For example, the chromosome for the default configuration is represented as the vector of values $\langle$50,6,0,0,20,12,0,0,1,0,1,0,0,1,0,0,1,0,0,0,1000,3,0,0$\rangle$, though this is merely one element of the configuration search space Ω. We wrote a decoding driver script for each clone detection tool. The driver converts the vector of values produced by EvaClone and CloudEvaClone to the required format used by each of the clone detection tools. Our genetic algorithm uses the tournament selection method to create mating pools. A single point crossover operator and a single point mutation operator is used to reproduce offsprings. The crossover rate is 0.8 and the mutation rate is 0.1.

4.3 Fitness Functions

The fitness function is another parameter to our approach and must be carefully chosen. For the empirical evaluation we focus on the agreement of clone detectors. As discussed, the more the clone detectors agree, the more trustable are their results. Thus, the goal is to find configurations that lead to maximal agreement of clone detectors.

Given a configuration, the fitness function returns a value indicating the degree of agreement of the clone tools on the subject systems for the given configuration. We distinguish between two use-cases for the fitness function, depending upon whether it is applied to a set of systems (seeking agreement on all of them with a *general* fitness) or a single subject system (with an *individual* fitness), in which case the clone tools should agree only on that specific system of interest:

Individual Task: Given a set of subject systems $SS = \{S_1, \ldots, S_m\}$, for a system $S_k \in SS$, search for a better configuration X for tool set TS containing n tools to get the maximum agreement of clone detection results on S_k. The *individual* fitness function for the *Individual Task* is:

$$f_I(TS(X), S_k) = \frac{\sum_{i=1}^{n}(i \times \text{AgreedLOC}[i])}{n \times \sum_{i=1}^{n} \text{AgreedLOC}[i]} \qquad (1)$$

General Task: Given a set of systems $SS = \{S_1, \ldots, S_m\}$, automatically search a better configuration X for tool set TS to get the maximum agreement of clone detection results on all the subjects in SS. The *general* fitness function for the *General Task* is based on Equation 1 and defined as:

Table 5: Best General and Individual Configurations Found by CloudEvaClone for the Clone Detection Tools

Tools	Parameter Name	Range	Configuration Settings										
			Default	General		Individual (Specific to Each Bellon Suite)							
				C Set	Java Set	*wel tab*	*cook*	*snns*	*psql*	*java doc*	*ant*	*jdt core*	*swing*
Tools	**Minimum Clone Size Settings (Applicable to Several Tools)**												
PMD, IClones, CCFinder	*MinToken*	10-300	50	14	26	30	10	13	13	31	29	29	26
ConQAT, Simian, NiCAD	*MinLine*	5-7	6	5	5	6	5	5	5	5	5	6	5
Tool	**Technique-Specific Configuration Settings**												
PMD	PMDIgnoreLiterals	0,1	0	0	1	0	1	1	0	1	1	1	1
	PMDIgnoreIdentifiers	0,1	0	0	1	1	1	0	0	1	1	1	1
IClones	MinBlock	0-300	20	8	6	0	6	9	7	4	3	5	6
CCFinder	TKS	1-300	12	10	7	4	8	8	9	1	4	1	3
Simian	ignoreCurlyBraces	0,1	0	0	1	0	1	1	1	0	1	1	0
	ignoreIdentifiers	0,1	0	0	1	0	0	0	0	1	1	1	1
	ignoreIdentifierCase	0,1	0	1	1	0	1	0	0	0	1	1	0
	ignoreStrings	0,1	0	1	1	1	1	1	1	1	1	1	1
	ignoreStringCase	0,1	1	1	0	0	1	1	1	1	0	0	0
	ignoreNumbers	0,1	0	1	0	1	1	0	1	1	1	0	1
	ignoreCharacters	0,1	0	1	1	1	1	1	1	1	0	1	0
	ignoreCharacterCase	0,1	1	1	0	1	1	1	1	0	0	1	0
	ignoreLiterals	0,1	0	0	0	0	0	1	0	0	0	0	0
	ignoreSubtypeNames	0,1	0	1	1	0	1	1	1	0	1	1	1
	ignoreModifiers	0,1	1	1	0	0	1	0	0	1	1	0	0
	ignoreVariableNames	0,1	0	1	0	1	1	1	1	0	0	1	1
	balanceParentheses	0,1	0	0	0	0	0	0	0	0	0	0	0
	balanceSquareBrackets	0,1	0	1	1	0	1	1	0	0	0	0	0
NiCAD	MaxLine	100-1000	1000	549	604	550	295	556	283	794	329	776	171
	UPI	0.0-0.3	0.3	0.3	0.2	0.3	0.3	0.3	0.3	0.2	0.3	0.2	0.2
	Blind	0,1	0	1	1	1	1	1	1	1	1	1	1
	Abstract	0-6	0	6	6	0	1	6	1	6	0	6	6

$$f_G(TS(X), SS) = \frac{1}{m} \sum_{S_k \in SS} f_I(TS(X), S_k) \qquad (2)$$

In these fitness equations AgreedLOC[i] represents the number of lines reported as cloned by exactly i tools.

For example, AgreedLOC[6] is the number of lines reported as cloned by 6 tools, AgreedLOC[1] represents the number of lines reported as cloned by only a single tool. Higher weight is given to those AgreedLOC[i] values with larger i, by multiplying AgreedLOC[i] by i. This ensures that the larger agreement on the clone detection results among the tools, the higher the fitness.

The individual fitness function is based on the following observation: We focus only on lines that are reported by at least one tool as cloned. If we draw a histogram that shows for each such line how many tools report this line as cloned, we want to maximise the area covered by the histogram. The sum of all AgreedLOC[i] is the number of lines reported as cloned by at least one tool. Thus, the denominator is the maximal achievable area. The numerator is the actual area covered. The general fitness function is computed from the individual fitness for each system; it is simply the average of the individual fitness functions.

4.4 Research Questions

We use a large-scale empirical study of CloudEvaClone to address the following research questions:

RQ1 (Default Agreement Baseline): How much agreement can be obtained using the default configuration of clone detection tools? We ask this question to establish the validity of our approach: If it turns out that the default configurations produce good agreement, then our approach would not be needed. We also need this result to provide a baseline against which to compare the results from our approach: How much better are they than the default configurations that would otherwise be used, were there no alternative.

RQ2 (Optimised General Agreement): How much agreement can our approach find among all tools using the general fitness function, which seeks to find agreement on all subject systems? This question establishes how useful our approach is at finding new default configurations for sets of tools and subject systems.

Of course, should it turn out that CloudEvaClone does, indeed, produce significantly better configurations than the currently used defaults, then, as a byproduct we shall also have a new default configuration for these tools.

RQ3 (Optimised Individual Agreement): How much agreement can our approach find among all tools using the individual fitness function, which seeks to find agreement on each individual subject system in isolation? If we can find even better configurations for individual systems, then this will be useful for researchers who wish to study the properties of clones arising from the systems studied themselves, rather than the detection techniques. By choosing a different configuration for each system, a researcher can reduce the confounding effect that a single configuration may have on different detection characteristics for different systems.

RQ4 (Accuracy): How much will recall and precision change when the optimised configurations are used? Clearly, a change in a tool's configuration will impact its recall and precision and usually there is a tradeoff between them. Configuration choices are intended to allow the user to prefer recall over precision and vice versa. It is thus important to know if the optimisation for agreement prefers recall or precision.

i	AgreedLOC[i]							
	weltab	*cook*	*snns*	*psql*	*javadoc*	*ant*	*jdtcore*	*swing*
6	6,941	2,156	6,197	6,598	725	227	11,002	5,774
5	909	2,094	7,409	4,523	448	296	5,222	4,001
4	452	3,214	6,161	6,690	574	459	6,853	7,769
3	355	3,773	4,706	7,501	470	519	6,878	5,558
2	393	7,964	9,231	17,822	800	832	11,714	9,780
1	937	9,996	13,225	28,014	2,124	3,976	25,657	33,007
Σ	9,987	29,197	46,929	71,148	5,141	6,309	67,326	65,889
f	84%	42%	51%	41%	45%	31%	47%	39%

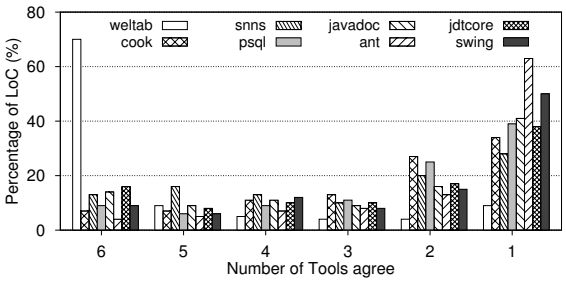

Figure 4: Default Configurations Achieve Poor Agreement on Clones

5. RESULTS AND DISCUSSION

While RQ1 can be answered by simply running the fitness evaluation component of EvaClone on a desktop, the experiments for RQ2 and RQ3 required 132,467 CPU hours. This would have taken approximately 15 years of continual execution on a conventional desktop machine. Therefore, we used CloudEvaClone to parallelise the computation and thereby render the execution time manageable. The significant computational effort involved is due to the need to evaluate our approach thoroughly. A future user of (Cloud)EvaClone would require only a tiny fraction of this computational effort to find suitable configurations.

5.1 RQ1: Default Agreement Baseline

To establish how much agreement can be obtained with the default configurations as our baseline, we compute the agreement and fitness values for the individual subject systems as shown in the table on the left-hand side of Figure 4. For each of the eight systems, the table shows the number of lines for which exactly i tools agree that the line is cloned. In addition, it shows the number of lines that at least one tool reports as cloned ($\sum$ AgreedLOC[i]) and the fitness value f.

For example, *weltab* has 6,941 out of 9,987 lines where all six tools agree that the line is cloned and 937 lines where only a single tool detects the line as cloned. Most of the time, all six tools agree for the *weltab* system and so, consequently, the fitness value f is high (0.84) for *weltab*. However, *weltab* is unusual; all other systems have much lower agreement (and fitness values between 0.31 and 0.51). In particular the number of lines where only one of the tools detected the line as cloned is very large for the other systems.

The histogram on the right of Figure 4 shows the agreement values from the table on the left as percentages of all lines reported as cloned. The figure confirms *weltab* as outlier and in most cases only one or two tools agree that a line is cloned. Thus, the answer to RQ1 is: **In their default configurations, clone detection tools have a low agreement on which lines are cloned.**

However, it could be the case that the configurations actually have a low impact on the results: perhaps the low agreement observed is due to the fundamentally different clone detection techniques the tools implement. To estimate the impact that the configurations *actually* have on the agreement, we compare randomly generated configurations with the default configuration.

For both sets of C and Java systems, 100 random (but valid) configurations are generated as a sample of the space of all valid configurations. Each generated configuration is applied to the six detection tools which are run on each of the eight subject systems.

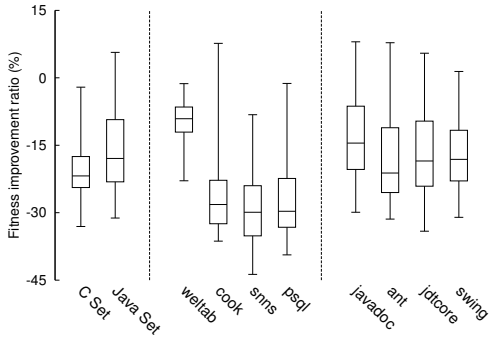

Figure 5: Default Configurations Perform Poorly

The fitness values of the generated configurations X are then compared with those of the default configuration X_d by computing their difference as a *fitness improvement ratio* of X to X_d:

$$fir(X) = \frac{f(TS(X), S) - f(TS(X_d), S)}{f(TS(X_d), S)} \quad (3)$$

In Equation 3, S is a set of subject systems (which could be a singleton set, in which case it applies to a single system). The equation can be used to asses fitness improvement ratios for both the general and the individual fitness functions.

Figure 5 shows the boxplots for the computed fitness improvement ratios. The two general sets (C and Java) are to the left of the figure, while the individual systems are to the right. As we can see from the figure, random configurations almost always cause a lower agreement than the default configuration, ranging up to almost 45% lower fitness values (*snns*). This shows that the agreement between clone detection tools is highly sensitive to their configurations. However, a small number of random configurations can actually improve the fitness (and thus increase the agreement), as can be seen for *cook* and the set of Java systems.

Therefore we conclude that, though default configurations favour agreement overall (compared to purely random configurations), there do, nevertheless, exist configurations that cause greater agreement between the results of clone detection tools than with the default configuration.

5.2 RQ2: Optimised General Agreement

Given that the default configurations cause low agreement and that randomly generated configurations can increase agreement, we use CloudEvaClone to search for configurations that maximise agreement. This experiment uses the *general* fitness function defined in Equation 2 to seek a configuration for the C and Java set that maximise agreement. In order to support inferential statistical testing, CloudEvaClone is executed 20 times. The fitness values for the

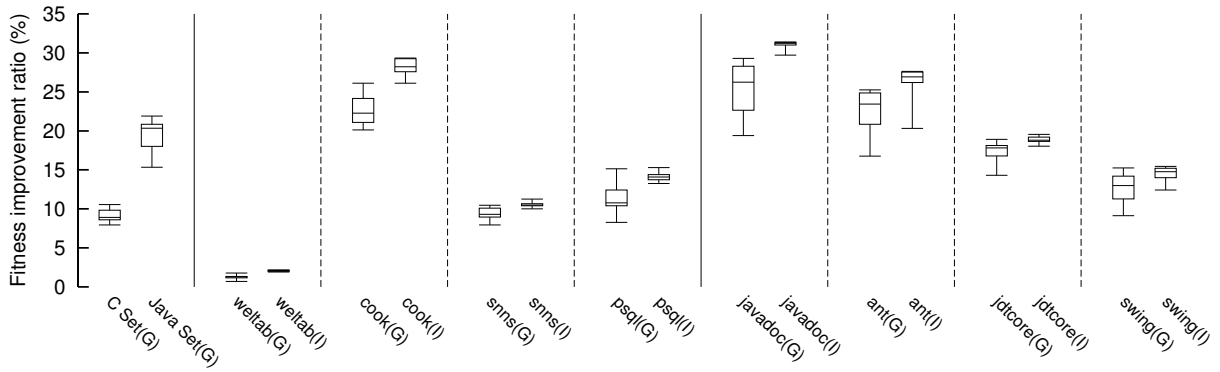

Figure 6: Agreement Improvement (over Defaults) for General and Individual Configurations

configuration generated by each run and each subject system are compared with those from the default configuration. The comparison is done based on the *fitness improvement ratio* (Equation 3) as was used to provide an answer to RQ1. Figure 6 shows the boxplots for the achieved improvements. The improvements of the general fitness for RQ2 are marked with a 'G', while the individual ones are marked with an 'I' (discussed with RQ3). The figure shows improvements for the Java (up to 21.9%) and C set (up to 10.6%) as well as for the individual systems. It is interesting to note that the improvements for the Java systems are larger than for the C systems. Moreover, *weltab* proves to be an outlier again.

To further confirm that the optimised configurations tend to have higher fitness values than default configurations, Wilcoxon signed rank tests were performed. For each system, 20 fitness value pairs are compared. Each pair consists of one fitness value for the general configuration from each GA run and the value for the default configuration. The null hypothesis is that the fitness values for the configurations we find are not significantly different from that of the default configuration. For all 8 systems the z-score was -3.92 and the p-value was 0.000 (three decimal places) conforming the observation from the box plots that our approach significantly outperforms the default configurations. In fact *every* one of the 20 executions of CloudEvaClone produces a better configuration. We also computed the effect size, r, in the standard way. That is, for a sample size N, the value of r is computed from the z-score: $r = abs(\frac{z}{\sqrt{N}})$. For all 8 systems, the effect size reported by this test is 'large' according to Cohen's effect size criteria [5] (which suggest 0.2 to 0.3 is a small effect, around 0.5 is a medium, while values above 0.8 denote large). Therefore, we answer RQ2 as follows: **CloudEvaClone finds configurations that are significantly better than the current default configurations and with a large effect size.**

The configurations found by the best of the 20 CloudEvaClone runs are reported in Table 5 (columns 5 and 6). It is interesting to observe the differences in configurations for the C and Java system sets which reflects the differences in coding styles and their impact on clone detection.

5.3 RQ3: Optimised Individual Agreement

By answering RQ2, new general configurations for C and Java systems have been established. As we have seen, there is a difference between C and Java systems—maybe there are even larger differences when CloudEvaClone searches for optimised configurations for the *individual* systems. RQ3 addresses this question using CloudEvaClone with the individual fitness function defined in Equation 1 and executing CloudEvaClone on each system in isolation to find optimised *individual* configurations.

Table 6: Individual Configurations Significantly Outperform the General Configurations ($\alpha = 0.05$)

Subjects	z-score	p-value	Significant?	r	Effect Size
weltab	-3.92	0.000	yes	0.88	large
cook	-3.92	0.000	yes	0.88	large
snns	-3.81	0.000	yes	0.85	large
psql	-3.70	0.000	yes	0.83	large
javadoc	-3.92	0.000	yes	0.88	large
ant	-2.80	0.005	yes	0.63	medium
jdtcore	-3.92	0.000	yes	0.88	large
swing	-2.65	0.008	yes	0.59	medium

Figure 6 shows the fitness improvements for the individual systems (I) side-by-side to the improvements previously achieved by the general configuration (G). The boxplots show that the individual optimisations not only lead to even greater agreement, but also that the range of improvements is smaller than for the general optimisation. Again, *weltab* is found to be an outlier.

To confirm that individual optimisations tend to generate higher fitness values than the general configurations, Wilcoxon signed rank tests are performed again. The null hypothesis is that fitness values of the general configurations are not significantly different from the ones for individual configurations. Table 6 shows the results are statistically significant in all cases, and with high effect sizes in all cases except *ant* and *swing*. Therefore, we answer RQ3 as follows: **CloudEvaClone can find even greater agreement using the individual fitness function applied to each subject system in isolation.**

To illustrate the improvements CloudEvaClone can provide, Figure 7 compares the fitness values and cumulative AgreedLOCs for the current default and CloudEvaClone-reported general and individual configurations for the largest two systems, *psql* and *swing*. In this figure, the columns located above the horizontal axis legend '≥ 5' denote the lines of code reported as cloned by at least 5 tools (i.e., AgreedLOC[5] + AgreedLOC[6]), while the columns above the legend '*potential cloned lines*' denote the number of lines detected as cloned by at least one detection tool (i.e. $\sum_{i=1}^{6}$ AgreedLOC[i]). As can be seen, the individual fitness gives the highest degree of agreement, while the currently used defaults offer the worst agreement, regardless of the number of tools from which we seek agreement.

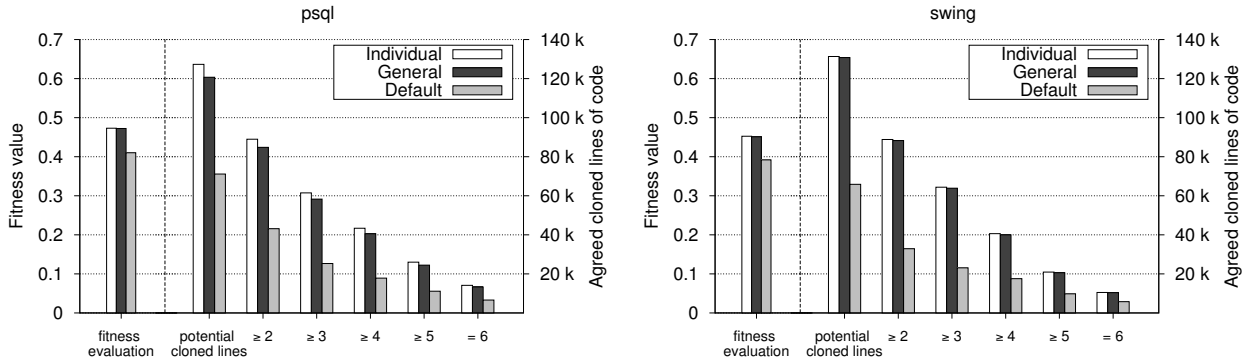

Figure 7: Agreement Levels for the Two Largest Systems *psql* and *swing*

5.4 RQ4: Accuracy

Whenever a new tool is introduced, the authors understandably want to investigate the performance of the new tool, hoping that it will outperform some competitor or state-of-the-art alternative tool or technique. In the literature [2, 34, 29], this is typically achieved using a 'gold standard' of human-assessed clones. The standard set that has emerged as a benchmark in many studies is the Bellon benchmark [2]. Using this benchmark, authors measure and compare the precision and recall of detection tools. As the tool's configurations allow them to favour recall over precision and vice versa, any comparison is only valid for one specific configuration. The optimisation for agreement has used the configuration space and it is important to know how this impacts precision and recall of the individual tools.

There are three accuracy criteria with which the impact of the optimised configurations can be studied:

1. The number of clone candidates: Higher agreement can be achieved by increasing the number of tools agreeing that a line is cloned or by increasing the number of tools agreeing that a line is not cloned. For the former one can expect more clone candidates and for the later one can expect fewer clone candidates.

2. Recall: If higher agreement is due to a larger number of clone candidates, then a higher recall can be expected.

3. Precision: If higher agreement is caused by fewer clone candidates, then a higher precision can be expected.

The first criteria can still be studied without a benchmark, but the other two criteria will use Bellon's benchmark which estimates precision and recall by comparing reported candidates against a (small number of) manually confirmed clone pairs. Bellon used two possible matchings of clone pairs to each other: 'Good' and 'OK' (see [2] for an explanation); we will only use the 'OK' matching.

Figure 8 shows the three criteria for the two largest systems *psql* and *swing*. Due to space restrictions, we only show results for two systems, but we will discuss the results for all systems. The first observation we can make is that the optimised configurations cause more candidates to be reported: For all eight systems and all five clone detectors, neither the individual nor the general optimised configurations have ever lead to a lower number of candidate pairs. In 46 out of 80 cases, the number of reported candidates more than doubled. Sometimes the number of reported candidate pairs exploded, as can be seen in Figure 8 for CCFinder and NiCAD on *psql* and for PMD on *swing*. The explosion is an artefact of the Bellon framework: If a tool reports a large clone class (i.e. a clone with many instances), then every pair of clones in that class will be reported as a candidate pair, leading to a quadratic explosion. The explosion is the reason why Simian has not been included in the final evaluation as the number of generated candidate pairs was too large to process.

With such an increase of candidate pairs, one expects an increase in recall and a drop in precision. This can generally be observed in the reported precision and recall as computed by the Bellon framework. There is one situation with a drop in recall: The Bellon framework reports a lower recall for IClones applied to *weltab* with the individual optimised configuration. The situation is different for precision and there are a few cases where the precision is increased: The individual optimised configuration causes a higher reported precision for PMD on *cook* and *psql*, for CCFinder on *snns*, and for NiCAD on *javadoc* and *swing*. The general optimised configuration causes a higher reported precision in the same cases except for PMD on *cook*. This can also be seen in Figure 8 for PMD on *psql* and for NiCAD on *swing*. Because the optimised configurations increase precision and recall at the same time, it may be the case that **the optimisation found better configurations** for PMD, CCFinder, and NiCAD. However, this may be purely coincidental due to the limitations of the Bellon benchmark.

Overall, one can answer RQ4 with **If CloudEvaClone is used to maximise agreement between clone detectors, recall will be favoured over precision and more candidates will be reported.** However, there may be situations where a higher precision is preferable over recall. In such situations one can simply choose a different fitness function as the fitness function is a parameter to (Cloud)EvaClone.

5.5 Actionable results from this work

We have seen that many empirical studies compare clone tools and techniques (Section 2). However, there remain important scientific concerns, expressed repeatedly in the literature, that current practices are potentially flawed due to the confounding configuration choice problem.

We have seen that default settings, widely used in previous empirical studies, offer a poor solution to the problem (RQ1) and that EvaClone can produce significantly better configuration choices. The primary actionable finding from this research is that future studies of clones can use (Cloud)EvaClone to find appropriate configurations for their studies.

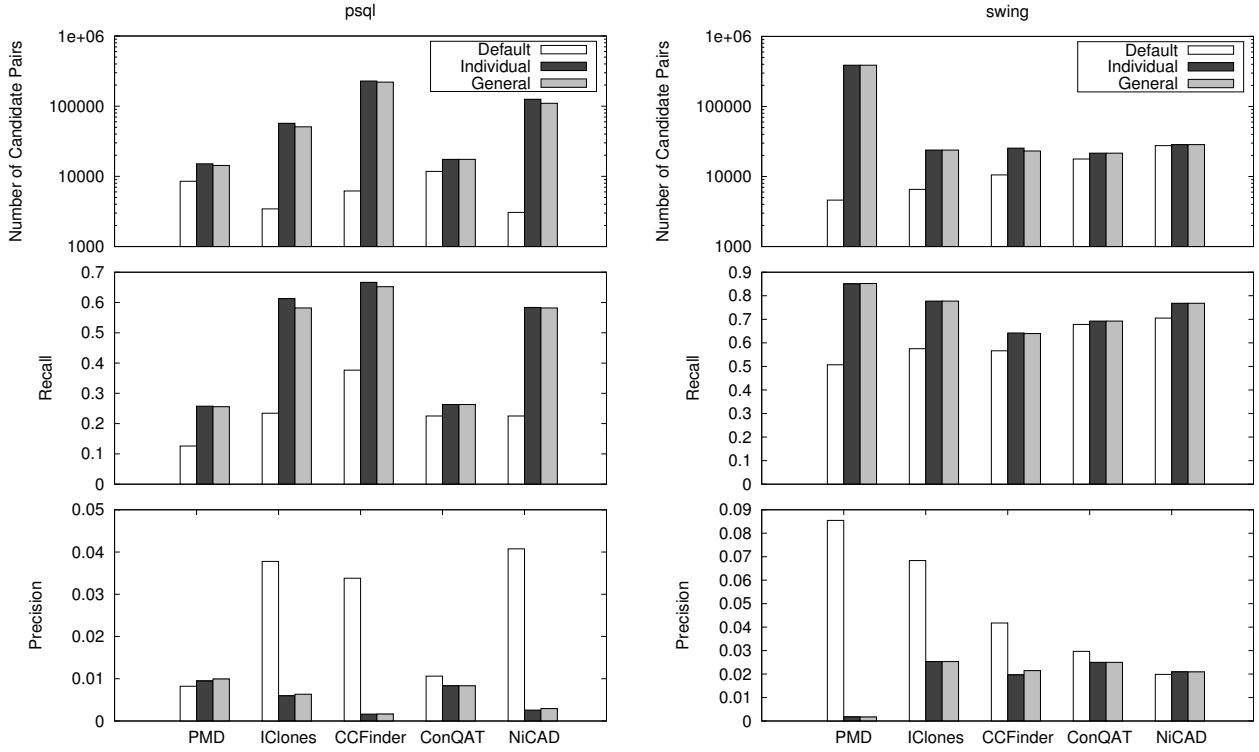

Figure 8: Results from applying Bellon's framework on *psql* and *swing*

EvaClone and CloudEvaClone, their source code, and all of the data used and reported in this paper can be found at the papers' companion website[1]. We will also release CloudEvaClone including our experimental environment as an Amazon Machine Image (AMI), which can be used to create a virtual machine within the Amazon Elastic Compute Cloud (EC2) thereby facilitating replication. Authors can also use EvaClone with a different fitness function to tune configuration choices for other problems (not simply those involving clone tool agreement as reported upon here).

5.6 Limitations and Threats to Validity

The subject systems used in this paper are all open source and not necessarily representative of all software systems. However, they do constitute a widely-used 'benchmark' set, so the results reported here have actionable consequences for the clone detection community.

The clone detection tools also cannot be regarded as representative of all clone detection tools: Though we show that our approach performs significantly better than the currently used default settings for these techniques, we cannot necessarily predict its effect on other techniques. However, we have used techniques that represent several different widely-used approaches to clone detection (including text-, token- and tree-based approaches). We therefore have some cause for confidence that CloudEvaClone may continue to prove useful for future, as yet unimplemented, tools and techniques. Note that the results are impacted by the approaches, e.g., including a PDG-based approach will change the outcome. Moreover, our fitness function favours recall over precision and is focussed on agreement – there may be situations where disagreement is more interesting.

[1] `http://www.cs.ucl.ac.uk/staff/Y.Jia/projects/eva_clone/`

Our approach to handling internal and construct validity is standard best practice for these forms of experiments [1, 13]: We have reported results for repeated runs, using non-parametric statistical testing for significance and have also reported results for the effect size. We set the alpha level (chance of a Type I error) at 0.05, which is widely regarded as a standard choice. We used a sample size of 20, which is guaranteed to be sufficient to avoid Type II errors in all experiments since all the Wilcoxon tests reported $p < 0.05$.

The results reported in RQ4 for recall and precision cannot be generalised due to the constraints of the Bellon framework. It would require an expert validating the candidates reported by each tool on each system for all three configurations (default, individual, general). With 3.4 million candidate pairs, even checking 1% of them is beyond being feasible.

6. CONCLUSION

We have introduced an approach to finding configurations for clone detection techniques that can be used to place empirical studies of clone detection on a more rigorous and firm footing. Our approach ameliorates the effects of the confounding configuration choice problem in clone studies. We demonstrated that this problem is an important one, widely recognised in the clone detection literature and for which there was, hitherto, no satisfactory solution.

We evaluated our approach with a large-scale empirical study, which revealed that it can find significantly better configurations than the defaults, and with a high effect size. We demonstrated that the current research practice of using defaults settings in empirical studies has to be changed. Our approach can be used to tune settings in clone detection experiments for specific subject systems.

7. REFERENCES

[1] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Intl. Conf. on Software Engineering*, 2011.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9), 2007.

[3] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan. An empirical study on inconsistent changes to code clones at the release level. *Sci. Comput. Program.*, 77(6), 2012.

[4] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.*, 31(10), 2005.

[5] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences (second ed.)*. Lawrence Erlbaum Associates, New Jersey, 1988.

[6] PMD's Copy/Paste Detector (CPD) 5.0, July 14 2012.

[7] E. Duala-Ekoko and M. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Trans. Softw. Eng. Methodol.*, 20(1), 2010.

[8] M. Funaro, D. Braga, A. Campi, and C. Ghezzi. A hybrid approach (syntactic and textual) to clone detection. In *Intl. Workshop on Software Clones*, 2010.

[9] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Intl. Conf. on Software Engineering*, 2008.

[10] N. Göde and R. Koschke. Incremental clone detection. In *European Conf. Software Maintenance and Reengineering*, 2009.

[11] J. Harder and N. Göde. Efficiently handling clone data: RCF and cyclone. In *Intl. Workshop on Software Clones*, 2011.

[12] M. Harman. The current state and future of search based software engineering. In *Future of Software Engineering*, 2007.

[13] M. Harman, P. McMinn, J. Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical software engineering and verification: LASER 2009–2010*. Springer, 2012.

[14] S. Harris. Simian 2.3.3, July 14 2012.

[15] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution? In *EVOL/IWPSE*, 2010.

[16] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Intl. Conf. on Software Engineering*, 2007.

[17] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective – a workbench for clone detection research. In *Intl. Conf. on Software Engineering*, 2009.

[18] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7), 2002.

[19] H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: memory comparison-based clone detector. In *Intl. Conf. on Software Engineering*, 2011.

[20] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE*, 2005.

[21] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Working Conf. on Reverse Engineering*, 2007.

[22] J. Krinke, N. Gold, Y. Jia, and D. Binkley. Cloning and copying between GNOME projects. In *Working Conf. on Mining Software Repositories*, 2010.

[23] J. Li and M. D. Ernst. CBCD: Cloned buggy code detector. In *Intl. Conf. on Software Engineering*, 2012.

[24] 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), 2006.

[25] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang. Improving trace accuracy through data-driven configuration and composition of tracing features. In *ESEC/FSE*, 2013.

[26] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the relation between changeability decay and the characteristics of clones and methods. In *Intl. Conf. on Automated Software Engineering – Workshops*, 2008.

[27] M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke, and K. A. Schneider. An empirical study of the impacts of clones in software maintenance. In *Intl. Conf. on Program Comprehension*, 2011.

[28] M. Mondal, C. K. Roy, S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *ACM Symposium on Applied Computing*, 2012.

[29] H. Nguyen, T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone management for evolving software. *IEEE Trans. Softw. Eng.*, 38(5), 2012.

[30] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms. In *Intl. Conf. on Software Engineering*, 2013.

[31] F. Rahman, C. Bird, and P. Devanbu. Clones: what is that smell? *Empirical Software Engineering*, 17(4-5), 2012.

[32] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software. In *Working Conf. on Reverse Engineering*, 2008.

[33] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Intl. Conf. on Program Comprehension*, 2008.

[34] 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), 2009.

[35] R. Tairas. Clone literature repository website, July 14 2012.

[36] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1), 2009.