Defending Against the Attack of the Micro-clones

Rijnard van Tonder, Claire Le Goues
Carnegie Mellon University
rvt@cmu.edu, clegoues@cs.cmu.edu

***Abstract*—Micro-clones are small pieces of redundant code, such as repeated subexpressions or statements. In this paper, we establish the considerations and value toward automated detection and removal of micro-clones at scale. We leverage the Boa software mining infrastructure to detect micro-clones in a data set containing 380,125 Java repositories, and yield thousands of instances where redundant code may be safely removed. By filtering our results to target popular Java projects on GitHub, we proceed to issue 43 pull requests that patch micro-clones. In summary, 95% of our patches to active GitHub repositories are merged rapidly (within 15 hours on average). Moreover, none of our patches were contested; they either constituted a real flaw, or have not been considered due to repository inactivity. Our results suggest that the detection and removal of micro-clones is valued by developers, can be automated at scale, and may be fixed with rapid turnaround times.**

I. Introduction

Developers often copy-paste code when programming [1]. This behavior sometimes results in large blocks of cloned code [2]. Other times, the repeated segment is quite small, sometimes referred to as a *micro-clone* [3]. Although this type of repetition can be benign, it can also have serious security implications. Consider the Apple "goto fail" vulnerability [4]:

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  goto fail;
  goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
  goto fail;
```

Fig. 1. Apple goto-fail vulnerability.

Here, the second `goto` statement is not bound by the `if` conditional; it is executed unconditionally and introduces a vulnerability that compromises the security of SSL/TLS. The fact that such a seemingly innocuous (and likely) copy-paste mistake could have such grave impact alarmed many.

Despite the fact that micro-clones are easy to recognize and are often amenable to straightforward fixes, they continue to persist in thousands of software artifacts, including high-profile codebases. Researchers in both academia and industry have produced effective solutions, including IDE integrations, to mitigate cloning during development [1, 5, 6]. However, there is a lack of open source tools that can address legacy micro-clones in open source repositories at scale. We observe that the combined advent of large-scale software mining infrastructures [7] and open source hosting sites such as GitHub, offers a key opportunity to solve this problem by detecting micro-clones at scale and then automatically patching them. We therefore investigate three broad questions with respect to legacy micro-clones in open source code:

1) Can micro-clones be effectively detected at scale?
2) To what extent can they be remediated automatically or pseudo-automatically?
3) Is the detection and removal of micro-clones valued by open source software maintainers?

The term "micro-clone" was introduced in previous work [3]. The premise of this previous work is orthogonal to ours, focusing on the "Last Line" effect,[1] but still suggested that micro-clones are "worth fixing" on the basis of 202 detections in 208 open source projects. We go further by presenting, to the best of our knowledge, the first principled investigation of detection and remediation of micro-clones at scale. Moreover, we analyze the results of suggesting patches to active repositories to inform a potentially automatic procedure. Our contributions:

- We present an open source solution that automatically detects micro-clones at scale. Our data set and results are two orders of magnitude larger than previous work [3].
- We deliver insights towards fully automatic repair of micro-clones in actively developed software, using our patch history serves as reference case study. For instance, 76% of our accepted patches trivially remove micro-clones (and do not require complex modification).
- We present an empirical evaluation of our approach to substantiate the value of detecting and fixing micro-clones. We made 43 meaningful contributions to 38 repositories. 95% of our contributions to active repositories were accepted rapidly, averaging a time of 15 hours from being requested to being merged.

II. Approach

A. Micro-clone Detection

Our first goal is to automatically detect micro-clones in real world software at scale. By definition, micro-clones are short *type-2 clones*, or "syntactically identical" expressions [2]. We implemented detection schemes for five micro-clone classes taken from an established commercial-based solution, PVS-Studio [8]. PVS-Studio is the only tool to our knowledge that distinctly classifies micro-clones. Our selection covers micro-clones at incremental granularities, from subexpressions to complete methods. Table I describes these classes, including real examples from our findings as illustration.

To demonstrate feasibility at scale, we leverage the Boa [7] software mining infrastructure, which provides a data set spanning millions of GitHub repositories, and a domain-specific language (DSL) that supports custom analyses over this data

[1] Micro-clones are most likely to occur in the last line/statement of a program.

TABLE I
MICRO-CLONE DESCRIPTIONS AND EXAMPLES.

Abbrev	PVS Code	Duplicate element	Example
`Binop`	(V501)	Binary operator subexpression.	`if(jsonSourceFileName != null && jsonSourceFileName != null){...}`
`CondExp`	(V517)	Conditional expression in an `if-else-if`.	`if (this.value < other.getValue())return -1;` `else if (this.value < other.getValue())return 0;`
`Stmt`	(V519)	Consecutive statements.	`lights = new Environment();` `lights = new Environment();`
`CondBody`	(V523)	Conditional bodies.	`if ( ... )return NumberFormat.getNumberInstance().format(rv);` `else if ( ... )return NumberFormat.getNumberInstance().format(rv);`

`Method`	(V524)	Method signature and body.	`static _EOTimer access000(_TimerThread x0){ return null; }`
			`...`
			`static _EOTimer access000(_TimerThread x0){ return null; }`

set. We studied the Boa *September 2015/GitHub* data set, which contains the parsed abstract syntax trees for all Java files in 380,125 Java repositories on GitHub. We use Boa's DSL to detect micro-clones by performing equivalence checking on ASTs. Our static analysis therefore comprises a collection of scripts in the Boa DSL, which we make available.[2] Since Boa is an open platform, users may customize or create additional scripts for detecting different types of micro-clones; an additional benefit over closed-source solutions [8].

Scope of detection. We implemented several checks to obtain greater precision in detecting micro-clones which are easily remediated: (1) We only consider the operators `|`, `&`, `||`, `&&`, `<`, `>`, `<=`, `>=` for `Binop` clones. (2) For `Stmt` clones, we do not consider duplicated assignment statements that contain the left-hand side variable in the right-hand side expression. (3) For `CondExp`, `Stmt`, and `CondBody`, we prioritize expressions that do not contain methods (which may have side-effects). (4) For `CondBody` and `Method`, we ignore empty conditional bodies.

B. Micro-clone Patches

Our foremost objective in constructing and issuing micro-clone patches is to assess whether software maintainers value such contributions. This lays groundwork for motivating an automatic patching procedure to remediate legacy micro-clones. Accepted patches thus inform analysis of which micro-clones are amenable to automatic patching, and speculate impact.

Our goals impose several considerations on our patching procedure. First, we must propose patches in a way that allows developers to correct instances of micro-clones rapidly, conveniently, and correctly. This sentiment draws on previous work [9], which substantiates that developers value non-intrusive patches if there are no drawbacks (including convenience and mental strain). Second, we sought to maximize the probability that issued pull requests would actually be reviewed. We therefore limited pull requests to popular repositories, measured by number of stars (user favorites). We heuristically elected a lower bound of 40 stars for Java repositories. Our patch workflow then proceeded as follows:

[2] https://github.com/kilida/boa-scripts

1) Sort Boa output of micro-clones by repository popularity, and determine whether the file still exists in the master branch through the GitHub web interface.
2) Consult the Boa output to determine the micro-clone location (note that Boa does not store line numbers in the AST representation). As an example, our `Binop` output pinpoints the position of a duplicated variable. The `Stmt` output indicates the duplicate variable name.
3) Use GitHub's web-interface to edit the file directly in-browser.

Step 1 was automatic; Step 2 was semi-automatic in pinpointing location. Step 3 required some minimal manual effort, as all patches reduce to one of two mechanical actions:

- REM: Delete the micro-clone, which preserves code semantics. In the `Binop` example of Table I, this corresponds to deleting the text `&& jsonSourceFileName != null`.
- MOD: Modify a micro-clone, depending on context. This action is sometimes required for the `Binop`, `CondExp`, and `Stmt` categories. Such changes were inferred from the context of the micro-clone or from developer suggestions, and in our cases amounted only to variable renaming.

Intuitively, REM patches only ever remove program elements based on duplicate nodes in the associated AST structure, and thus can be straightforwardly automated. We present additional insight and analysis of REM and MOD patches, along with examples, in Section III. Finally, we emphasize that all of our patches were issued solely through GitHub's web-interface, a testament to the simplicity of fixing micro-clones.

III. RESULTS

A. Micro-clone Detection

Our analysis detects thousands of instances of micro-clones, summarized in the first three columns of Table II. We ran our Boa scripts over 380,125 Java repositories; all completed within one hour. We therefore expect our analysis to scale well to any particular project. Each detected micro-clone is unique. We also list the number of unique repositories in which we detected clones, since some repositories introduce many uninteresting clones. For example, a single repository accounts for 264 entries in the `Method` category, due to duplicate files.

While all entries are accurate in the sense that they truly

identify duplicated code, not all correspond to a fixable flaw. Some micro-clones are intentional, occurring in test cases. For example, we ran into some test cases for clone detection tools. Thus, our results present a conservative overapproximation of the fixable micro-clones that we wish to remediate in practice.

TABLE II
MICRO-CLONE DETECTION AND PATCH RESULTS.

	Micro-clones		*Pull Requests*			
		Unique	Total		Merged	
Type	**Count**	**Repos**	REM	MOD	REM	MOD
`Binop`	4,825	2,204	9	7	7	3
`CondExp`	1,412	1,354	2	2	-	1
`Stmt`	4,281	3,182	14	3	6	1
`CondBody`	13,296	8,351	5	1	3	-
`Method`	490	103	-	-	-	-
Total	24,304	12,141	30	13	16	5

The `CondBody` category dominates the results, due to a common coding idiom we observed in many projects. Consider the following example from our results:

```
1  if (inspector.getGetterMethods().get(fieldName) != null) {
2    // field without setter
3    return null;
4  } else {  // public field
5    return null;
6  }
```

Fig. 2. A Repeated Conditional Body.

The programmer *wants* the same behavior to be executed under various conditions, but explicitly enumerates those conditions rather than combining them into a single conditional, to support their mental model of the code. As Figure 2 illustrates, we noted that this idiom is often accompanied by clarifying comments. Such instances are correctly flagged as micro-clones and could be automatically corrected in a semantics-preserving way. However, this compiler-like optimization comes at the risk of disrupting the programmer's mental model of the code. We did not propose this type of intrusive corrections unless the conditional expressions were also duplicated.

By contrast, our check for equivalent methods (`Method`) did not reveal many entries. Although we found micro-clones that could be patched, none occurred in active, popular repositories. Method stubs are a common cause of duplication; false positives include equivalent methods that are part of test cases.

B. Micro-clone Patches

Using the Boa output, we issued pull requests to every open source project on GitHub that met our criteria of being active and popular. We issued 43 pull requests to 38 unique repositories. All repositories to which we submitted pull requests lie in the upper 92nd percentile of Java repositories with one or more star, and in the upper 99th percentile of Java repositories with zero or more stars. The most popular repository we contributed to, `libgdx`, is ranked 22nd on GitHub with 8,584 stars, and 10th in the number of user forks. The lower bound of the percentile is determined by a project with 46 stars. Table II summarizes the result of patches: 21 pull requests were accepted. Investigating the 22 unmerged pull requests, we note that the implicated repositories had low levels of activity: none of them had merged commits or pull requests for at least two weeks prior to our interation. For some, months had passed since the most recent pull request was merged. We thus ascribe the lack of review of many of our unmerged pull requests to temporary or permanent inactivity. Under this criterion, we consider 18 projects of 38 to be active, having had a commit or pull request within the recent two weeks of our pull request submission.

Sentiment Toward Patches. Assuming an accurate, fully automated solution to micro-clones, we want to know whether software maintainers value such contributions to their codebases. Our results support the affirmative: 20 of 21 patches to 18 unique active repositories were accepted. All patches were merged within 15 hours, on average. Patch #21 is an outlier, and was accepted to an additional repository after three days.

Micro-clone patches exhibit attractive attributes that positively distinguish them from other pull requests. First, they appear more likely to be merged than larger pull requests: 14 out of 18 repositories have pull requests which maintainers are not willing to merge, yet our changes were accepted. Importantly, we observed no cases in which another open pull request was merged but ours was not. Moreover, some contribution guidelines require opening a JIRA ticket before submitting a pull request. In light of micro-clones, this prompted one maintainer to ask "Do we really need JIRAs even for such simple and obvious changes?". We observed that for the four repositories stipulating this guideline, the requirement for us to open a ticket was lifted.

Ten developers expressed gratitude after merging our pull requests, including comments such as "Nice find!". The rest were merged silently. Three pull requests contained developer comments suggesting a variable renaming. In summary, all pull requests were acknowledged as real issues, and all but one were rapidly integrated into the codebase.

Patch Distribution. In assessing our ability to automate patches, we considered the actions required to construct patches, whether REM or MOD. Our results are summarized in Table II. With respect to the REM column of merged pull requests, we issued 19 patches that trivially remove micro-clones. However, three of these instances in fact required a MOD action, based on developer feedback. In these cases, developers suggested a renaming of variables, much like the fix in Figure 3, instead of removal. Overall, the majority of our pull requests (76%) are in the REM category, and were merged without any interaction.

Qualitative Considerations. Based on our patch history, we qualitatively observe several traits of micro-clones that have bearing toward fully automated patching. First, the most obvious and trivial occurrences are duplicated statements `Stmt` with Java primitives on the right hand side. Since primitives have no side-effects, (i.e. duplication does not have semantic

implications), such statements can be detected and removed entirely automatically.

For subexpressions (BINOP), checks against boolean primitives, `null`, enum constants, and `instanceof` are likely candidates for automatic removal. However, subexpressions containing methods (which may have side effects), or even checks on integer types raise a complication. For instance, consider the following patch we issued:

```
1   -  if(i1 == -1 || i1 == -1) {
2   +  if(i1 == -1 || i2 == -1) {
3        i = Math.max(i1, i2);
4      }
```

Fig. 3. A Simple Micro-clone Patch.

A human-in-the-loop can easily infer the change from `i1` to `i2` based on the use of `i2` in line 3. Even for simple integer comparisons, additional reasoning on variable usage is required in an automated approach. Although removing the additional subexpression is clearly semantics-preserving, doing so does not capture the desired change. Nevertheless, a promising result is that the cases which we have identified as being most amenable to automated patching (`Binop` and `Stmt`) also constitute the largest portion of feasible pull requests.

IV. Threats and Limitations

Large-scale detection of micro-clones is susceptible to producing false positives not easily distinguishable from real issues. For example, we found test cases that polluted our raw results. However, anecdotally, our heuristics address this issue.

Pull requests on social hosting sites can be valued for their contribution to the community spirit. A risk to the validity of our observations on the value of our patches based on our interactions with GitHub repositories is that they were accepted for social rather than technical reasons. However, we observe that maintainers of the repositories we targeted use discretion in considering pull requests: only four of the 18 repositories which merged our patches had *no* outstanding pull requests. Moreover, we observed that developers may suggest fixing changes based on our MOD actions. Hence, thought was given to proposed patches and not blindly accepted.

Some limitations to our solution are imposed by the Boa infrastructure. Boa operates over a snapshot of GitHub repositories from 2015. Inaccuracies in the analysis may result purely because code, files, or projects have changed. However, this limitation does not undermine our approach in terms of scalability. Second, the Boa infrastructure currently only parses Java. Boa's language-agnostic DSL is promising for extending our analysis to other languages such as C or Python, were ASTs available; we have no reason to believe that our approach would not generalize accordingly.

V. Conclusion

We presented a solution for detecting micro-clones at scale, and established the utility of the results for automatically remediating them. Using Boa, we found thousands of valid micro-clones in 380,125 Java repositories. Furthermore, we made 43 meaningful contributions to 38 repositories, and observed a rapid acceptance rate of 95% for pull requests issued to active repositories. Our results suggest that micro-clones can be detected effectively at scale, and that this capability can be leveraged to facilitate rapid, automatic removal of legacy micro-clones which continue to persist in high-profile software.

VI. Acknowledgements

This research was funded in part by the Air Force under Contract #FA8750-15-2-0075 and by the US Department of Defense through the Systems Engineering Research Center (SERC), Contract H98230-08-D-0171. Any opinions, findings, or recommendations expressed are those of the authors and do not necessarily reflect those of the US Government.

References

[1] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Predicting consistency-maintenance requirement of code clones at copy-and-paste time," *IEEE Transactions on Software Engineering*, vol. 40, no. 8, pp. 773–794, 2014.

[2] 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.*, vol. 74, pp. 470–495, 2009.

[3] M. Beller, A. Zaidman, and A. Karpov, "The last line effect," in *International Conference on Program Comprehension*, ser. ICPC '15, 2015, pp. 240–243.

[4] Adam Langley, "Apple's SSL/TLS bug," https://www.imperialviolet.org/2014/02/22/applebug.html, 2014, online; accessed 3/26/16.

[5] Microsoft, "Finding Duplicate Code by using Code Clone Detection," https://msdn.microsoft.com/en-us/library/hh205279.aspx, 2015, online; accessed 3/26/16.

[6] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, "Detecting differences across multiple instances of code clones," in *International Conference on Software Engineering*, ser. ICSE '14, 2014, pp. 164–174.

[7] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *International Conference on Software Engineering*, 2013, pp. 422–431.

[8] "PVS-Studio," http://www.viva64.com/en/pvs-studio/, 2016, online; accessed 3/26/16.

[9] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: detecting and fixing performance problems that have non-intrusive fixes," in *International Conference on Software Engineering*, 2015, pp. 902–912.

[10] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *International Conference on Software Engineering*, ser. ICSE '07, 2007, pp. 96–105.

[11] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Empirical Software Engineering*, vol. 15, no. 1, pp. 1–34, 2010.