Empir Software Eng
DOI 10.1007/s10664-016-9489-6

The last line effect explained

Moritz Beller[1] · Andy Zaidman[1] · Andrey Karpov[2] · Rolf A. Zwaan[3]

Abstract Micro-clones are tiny duplicated pieces of code; they typically comprise only few statements or lines. In this paper, we study the "Last Line Effect," the phenomenon that the last line or statement in a micro-clone is much more likely to contain an error than the previous lines or statements. We do this by analyzing 219 open source projects and reporting on 263 faulty micro-clones and interviewing six authors of real-world faulty micro-clones. In an interdisciplinary collaboration, we examine the underlying psychological mechanisms for the presence of these relatively trivial errors. Based on the interviews and further technical analyses, we suggest that so-called "action slips" play a pivotal role for the existence of the last line effect: Developers' attention shifts away at the end of a micro-clone creation task due to noise and the routine nature of the task. Moreover, all micro-clones whose origin we could determine were introduced in unusually large commits. Practitioners benefit from this knowledge twofold: 1) They can spot situations in which they are likely to introduce a faulty micro-clone and 2) they can use PVS-Studio, our automated micro-clone detector, to help find erroneous micro-clones.

Communicated by: Christian Bird and Rocco Oliveto

✉ Moritz Beller
m.m.beller@tudelft.nl

Andy Zaidman
a.e.zaidman@tudelft.nl

Andrey Karpov
karpov@viva64.com

Rolf A. Zwaan
zwaan@fsw.eur.nl

[1] Delft University of Technology, Delft, The Netherlands

[2] OOO "Program Verification Systems," Tula, Russian Federation

[3] Department of Psychology, Erasmus University Rotterdam, Rotterdam, The Netherlands

Published online: 29 December 2016

Keywords Micro-clones · Code clones · Clone detection · Last line effect · Psychology · Interdisciplinary work

1 Introduction

Software developers oft need to repeat one particular line of code several times in succession with only small alterations, like in this example from TrinityCore:[1]

Example 1 TrinityCore

```
1 x += other.x;
2 y += other.y;
3 z += other.y;
```

The 3D-coordinates of the `other` object are added onto the member variables representing the coordinates `x, y, z`. However, the last line in this block of three similar lines contains an error, as it adds the `y` coordinate onto the `z` coordinate. Instead, the last line should be

```
3 z += other.z;
```

Another example from the popular web browser Chromium[2] shows that this effect also occurs in similar statements within one single line:

Example 2 Chromium

```
1 std::string host = ...;
2 std::string port_str = ...;
3 if (host != buzz::STR_EMPTY && host != buzz::STR_EMPTY)
```

Instead of comparing twice that `host` does not equate the empty string, in the last position, `port_str` should have been compared:

```
3 if (host != buzz::STR_EMPTY && port_str != buzz::STR_EMPTY)
```

Lines 1–3 from Example 1 are similar to each other, as are the statements in the if clause in line 3 from Example 2. We call such an extremely short block of almost identically looking repeated lines or statements a *micro-clone*. Through our experience as software engineers and software quality consultants, we had the intuition that *the last line or statement in a micro-clone is much more likely to contain an error than the previous lines or statements*. The aim of this paper is to verify whether our intuition is indeed true, leading to two research questions:

RQ 1 Is the last line in a multi-line micro-clone more likely to contain an error?
RQ 2 Is the last statement in a single-line micro-clone more likely to contain an error?

As recurring micro-clones are common to most programming languages, the presence of a last line effect can impact almost every programmer. If we can prove that the last of a series of similar statements is more likely to be faulty, code authors and reviewers alike will

[1]TrinityCore is a popular open-source framework for the creation of Massively Multiplayer Online Games (MMOGs), www.trinitycore.org.

[2]Chromium is the open-source part of Google Chrome, www.chromium.org.

know which code segments to give extra attention to. This can increase software quality by reducing the amount of errors in a program.

One natural way to come up with code like in Example 1 and 2 is to copy-and-paste it. By closely examining the origin of micro-clone instances, we come to the conclusion that developers employ a variety of different mechanical patterns to create micro-clones, most important among which is copy-and-pasting on a line-per-line basis. Copy-and-pasting and cloning are some of the most widely used idioms in the development of software (Kim et al. 2004). They are easy and fast to do, hence cheap, and the copied code is already known to work. Though often considered harmful (Kapser and Godfrey 2008), sometimes (micro-) cloning is in fact the only way to achieve a certain program behavior, like in the examples above. A number of clone detection tools have been developed to find and possibly remove code clones (Bellon et al. 2007; Roy et al. 2009). While these automated clone detection tools have produced very strong results down to the method level, they are ill-suited for recognizing micro-clones in practice because of an abundance of false-positives.

When we posted a popular science blog entry[3] about the last line effect, it was picked up quickly and excitedly in Internet fora.[4] Many programmers agreed to our observation, often assuming a psychological reason to cause the effect. This leads to our last research question:

RQ 3 What are the reasons for the existence of faulty micro-clones and the last line effect in particular?

Through interviews, deeper technical analyses and interdisciplinary work with a psychologist, we research whether and which psychological aspects could play a role in the last line effect. We first collect phenomena from the well-established area of cognitive psychology and then research if they explain the last line effect in micro-clones.

By building upon our initial investigation of the last line effect (Beller et al. 2015), we make the following contributions:

- We define and introduce the term *micro-clone*.
- We introduce techniques for the detection of faulty micro-clones implemented in the automated static analysis tool (ASAT, Beller 2016) PVS-Studio, which cannot be found with traditional clone detection.
- We manually investigate the error proneness of 263 micro-clones in 219 well-known open-source systems (OSS), based on a total of 1,891 warnings.
- We provide an initial analysis of the underlying psychological mechanisms behind the existence of the last line effect.
- We lead six interviews with authors of micro-clones in real-world systems.
- We conduct a repository analysis on four well-known OSS projects based on the results of the interviews, investigating abnormally large commit sizes.

Our findings show that in micro-clones similar to Examples 1 and 2, the last line or statement is significantly more likely to contain an error than any other preceding line or statement. Rather than technical complexity of the micro-clone, psychological reasons seem to be the dominant factor for the existence of these faulty micro-clones, mostly related to working memory overload of programmers. An initial investigation with five projects reveals that all micro-clones were introduced in abnormally large commits at often unusual

[3] www.viva64.com/en/b/0260

[4] www.reddit.com/r/programming/comments/270orx/the_last_line_effect

work hours. This knowledge and our ASAT PVS-Studio can support human programmers in reducing the amount of simple last line-type of errors they commit by automating the detection of such faulty pieces of code.

2 Study Setup

Our study consists of two empirical studies C_1 and C_2. In this section, we describe how we set-up the two empirical studies on micro-clones and which study objects we selected.

2.1 Study Design C_1: Spread and Prevalence of the Last Line Effect within Micro-Clones

Study C_1 examines how wide-spread the last line effect is within micro-clones. We statistically examine the existence of the last line effect within micro-clones in five easily replicable steps. Moreover, in an effort to shed light on how developers create them, we added an analysis on the origin and destination of micro-clone instances.

1. Run the static analysis checker PVS-Studio on our study objects, with all checks enabled. PVS-Studio is a commercial static analysis tool developed by the Russia-based company “OOO Program Verification Systems” and incorporates dozens of static analyses from detecting clones to recognizing anti-patterns of using specific library functions in C. For replication purposes, a free trial of PVS-Studio is publicly available.[5]
2. Inspect the corpus of warnings from PVS-Studio and remove false-positives and warnings not related to micro-cloning.
3. For each faulty micro-clone, count the total number of lines (RQ 1) or statements (RQ 2) and denote which lines or statements are faulty. If possible, infer the likely origin and destination of the micro-clone (for example, in Example 6, the origin would be line 2 and the destination line 3).
4. Naively, we assume each line has the same likelihood $1/n$ of containing an error (H_0), independent of its position in an n-line long faulty micro-clone. For example, lines 1 and 2 in a 2-liner clone each have a 0.5 probability of containing an error. However, if we can show that the error distribution per line from step (3) significantly differs from such a uniform distribution on a $\sigma = 0.05$ significance level, we reject H_0 and assume a non-uniform error distribution. For each micro-clone length n, we use Pearson’s χ^2 test with $n - 1$ degrees of freedom to compare our empirical distribution’s goodness-of-fit to a $1/n$-equipartition.
5. If the test in step 4 finds a significant difference between the two distributions, we calculate the odds ratio between them as an intuitive measure of how strong the effect size of the last line effect is Bland and Altman (2000).

2.2 Study Design C_2: Analyzing Reasons Behind the Existence of the Last Line Effect

Having established the existence of the last line effect in C_1, we want to gain insight into the reasons why it exists (RQ 3). To this aim, we build an initial theory based on related work in the domain of cognitive psychology together with Rolf Zwaan, professor of cognitive

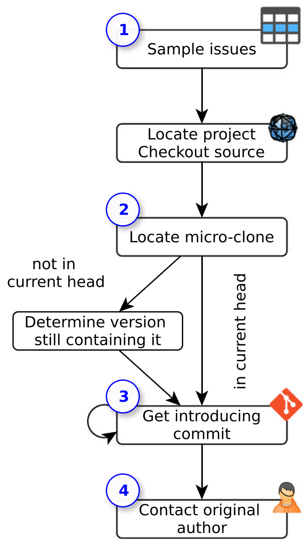
[5]www.viva64.com/en/pvs-studio-download

psychology. To obtain anecdotal evidence on micro-clones through interviewing developers and to further corroborate these cognitive explanations, in study C_2, we interview developers that authored last line effect instances. We specifically interviewed developers who authored micro-clones that we found in C_1. The emerging observations will aid us in creating an initial psychological explanation. By only contacting developers who we knew had created a micro-clone, we make our interviews (1) more focused on a concrete instance of the phenomenon that our interviewees could personally relate to and (2) more relevant by approaching an audience that we could prove had authored a faulty micro-clone in the past.

Figure 1 depicts our general study design. It centers around finding and contacting the original author of a micro-clone which in many cases is not present in the project's latest checkout anymore. The design comprises four primary steps:

1. We randomly sample projects and micro-clones to investigate from these projects, since carrying out C_2 is a tedious manual process that involves contacting and interviewing developers. Assuming a standard response rate for cold calling surveys of 30 %, the resulting three interviews would likely give us sufficient information to guide the creation of our initial psycho-cognitive explanation. For each micro-clone, we have to familiarize us with the project's development guidelines and check out their repository.
2. Next, we locate the micro-clone in the source tree of the project. Since many projects fixed our observations in the meantime and the clone is then not present in the current head, this step requires different search strategies: we start by checking out the repository at the date of our inspection for C_1. If this fails, for example because the code around the micro-clone was refactored (or the history force-overwritten), we try to track down the commit that removed the micro-clone by searching the project's issue tracker. If all else fails, we resort to a full text search (via `ag`) in every commit of the project's history.
3. Once we tracked down the original micro-clone, we follow its history, using `git blame`, to ensure we receive both refactorings that were applied to it as well as its true original author.

Fig. 1 Study design of C_2



4. In the last step, we use `git blame -e` to obtain the developers' email addresses to contact them. In an attempt to maximize the response rate, we also perform a web search to acquire additional personal information about the developers and verify the timeliness of the contact email addresses. We also made clear we will not disclose their identity to incentivize honest answers. We then send short personalized emails containing the micro-clone they authored, how it was later modified or fixed, the context of the bug, why we do the investigation, and a set of questions on the micro-clone at hand.

2.3 Study Objects

To ensure the replicability and feasibility of our study, we focused on well-known open-source systems. Among the 219 OSS we studied in C_1, we found instances of defective micro-clones in such renowned projects as the music editing software Audacity (1 finding), the web browsers Chromium (9) and Firefox (9), the XML library libxml (1), the databases MySQL (1) and MongoDB (1), the C compiler clang (14), the ego-shooters Quake III (3) and Unreal 4 (25), the rendering software Blender (4), the 3D visualization toolkit VTK (8), the network protocols Samba (4) and OpenSSL (2), the video editor VirtualDub (3), and the programming language PHP (1). For C_2, we sampled 10 micro-clones from the projects Chromium, libjingle, Mesa 3D, and LibreOffice.

2.4 How to Replicate This Study

To foster replication of this study, we have made the complete data set and all analyses available as a replication package.[6] The package includes all un-filtered warnings from PVS-Studio, separated into the older data used for our ICPC paper (Beller et al. 2015) `findings_old/` and the newer data added for this paper `findings_new/`. Moreover, it contains the analyzed and categorized micro-clones (`analyzed_data.csv`) together with an evaluation spreadsheet (`evaluation.ods`) and the results from the repository analysis of C_1 and C_2. We also provide the R scripts to replicate the results and graphs in this paper. Finally, we share a draft of the questions we sent to developers.

3 Methods

In this section, we outline traditional clone detection, why it is ill-suited for the recognition of micro-clones, and how we circumvented this problem with our tail-made static checks, our origin inference of micro-clone instances and commit size analysis.

3.1 Why Current Clone Detectors are not Suitable

As Examples 1 and 2 demonstrate, the code blocks that we study in this paper are either textually identical or contain "syntactically identical cop[ies]; only variable, type, or function identifiers have [...] changed." (Koschke 2007) This makes them *extremely short type-1 or type-2 clones* (usually shorter than 5 lines or statements), which we refer to as *micro-clones*.

[6]http://dx.doi.org/10.6084/m9.figshare.1313697

Traditional code clone detection works with a token-, line-, abstract syntax tree (AST), or graph-based comparison (Koschke 2007). However, to reduce the number of false-positives, all approaches are in need of specifying a minimal code clone length for their unit of measurement (be it tokens, statements, lines or AST nodes) when applied in practice. This minimal clone length is usually in the range of 5-10 units (Bellon et al. 2007; Juergens et al. 2009), which makes it too long to detect our micro-clones of length 2 to 5 units.

Consider Example 1, in which all lines 1-3 together form the micro-clone class. There are three micro-clones of this class, since each line 1-3 represents a single instance. Every micro-clone instance in Example 1 consists of a variable, an assignment operator and the assigning object and its member variable, so its length in abstracted units is four.[7]

3.2 How We Found Faulty Micro-Clones Instead

As clone detection is not able to reliably detect micro-clones in practice, we devised a different strategy to find them. Our research questions aim not at finding all possible micro-clones, but only the ones which are faulty. With this additional constraint, we could design and implement a handful of powerful analyses based on simple textual identity. These are able to find faulty code that is very likely the result of a micro-clone. Table 1 lists and describes the twelve analyses that found micro-clones in our study.[8] The last column summarizes the within- and multi-line code clones onto the number of all warnings found for this error code. For example, the analysis V501 simply evaluates whether there are identical expressions next to certain logical operators. If so, these are at best redundant and therefore cause a maintenance problem, or at worst, represent an actual bug in the system. Other analyses are not as specific toward micro-cloning as V501. We inspected all 526 warnings manually and only included the 272 containing an actual micro-clone in our study. Table 1 also shows that 78 % of our micro-clones stem from one analysis only (V501), which has a very low false-positive rate of 97 %. Other analyses have a higher likelihood of not only being triggered by micro-clones.

3.3 How We Inferred the Origin of an Erroneous Micro-Clone Instance

To be able to make qualified statements about why a last line effect might exist in RQ 3, we additionally identify, for each micro-clone class, the copied erroneous clone instance and the instance it likely originated from. While this a-posteriori analysis cannot provide us with absolute certainty that the clones were created in this way, we have convincing evidence that at least some developers mechanically create micro-clones this way (see RQ 3). Like in Example 1, in the majority of cases, it is most often immediately clear which is the influencing and which the influenced unit in a micro-clone: The erroneous line 3 contains left-over fragments from line 2, implying an influence from 2 (origin) to 3 (destination). Most micro-clones exhibit a similar natural order that determines origin and destination, either lexicographically like `x`, `y`, `z` in Example 1 or cardinally:

[7]Some clone detectors would count the assigning object and the reference to the member variables as one unit. Following this definition, the length in units would be three.

[8]For a more detailed description of the analyses, refer to http://viva64.com/en/d/0368/.

Table 1 Error Types from PVS-Studio and Their Distribution in our 219 OSS systems

PVS Error Code	Description	#within line clones	#multi line clones	Σ#/All
V501	There are identical sub-expressions to the left and to the right of the `foo` operator.	104	108	212/217
V517	The use of `if (A) {...} else if (A) {...}` pattern was detected. There is a probability of logical error presence.	0	8	8/58
V519	The `x` variable is assigned values twice successively. Perhaps this is a mistake.	0	23	23/117
V523	The `then` statement is equivalent to the `else` statement.	0	5	5/47
V524	It is odd that the body of `Foo_1` function is fully equivalent to the body of `Foo_2`.	0	3	3/13
V525	The code containing the collection of similar blocks. Check items `X, Y, Z, ...` in lines N1, N2, N3, ...	1	1	2/11
V537	Consider reviewing the correctness of `X` item's usage.	0	8	8/10
V570	The variable is assigned to itself.	1	1	2/17
V571	Recurring check. This condition was already verified in previous line.	0	2	2/17
V581	The conditional expressions of the `if` operators situated alongside each other are identical.	0	2	2/13
V583	The `?:` operator, regardless of its conditional expression, always returns one and the same value.	0	1	1/7
V656	Variables are initialized through the call to the same function. It's probably an error or un-optimized code.	0	4	4/8
Σ		106	166	272/535

Example 3 cmake

```
p[0] = 0xfc | ((wc >> 30) & 0x01);
p[1] = 0x80 | ((wc >> 24) & 0x3f);
p[1] = 0x80 | ((wc >> 18) & 0x3f);
p[2] = 0x80 | ((wc >> 12) & 0x3f);
```

Even in cases where there is no explicit natural order as in Examples 1 and 3, the code context often motivates an implicit order, like in Example 2: It would be against the order of their previous definitions to put `port_str` in the first place and `host` in the second place in line 3. Hence, we assume that the first instance of the micro-clone `host != buzz::STR_EMPTY` is the influencing origin and the second instance is the destination.

The general problem when reasoning about the origin and destination of micro-clones in our data set is 1) the possible variable clone length and 2) the expected relatively fewer micro-clones of length greater than 4. In order to be able to generalize over different micro-clone lengths nonetheless, we calculate, for each micro-clone i, $\delta_i = \text{line}_i(Destination) - \text{line}_i(Origin)$, resulting in the proximity distribution $\Delta_{Dest-Orig}$.

A value of 1 indicates an inference from the immediately preceding unit, as in Example 4. A value of 0 denotes that the error occurred within the same micro-clone instance. A value of -1 denotes a swapped pair of clones, in which the second influenced the first:

Example 4 UnrealEngine4

```
1  return cy().isRelative()
2      || cy().isRelative()
3      || r().isRelative()
4      || fx().isRelative()
5      || fy().isRelative();
```

Here, we would have expected `cx().isRelative` in line 1, instead of `cy().isRelative`, which seems to be influenced by the second line. Natural order, as well as lines 3 and 4 suggest that the micro-clone start with `return cx().isRelative()` in line 1 instead.

Hence, adding up the number of values where $\Delta_{Dest-Orig} = 1$ or $\Delta_{Dest-Orig} = -1$ gives us the number of clones that are direct neighbors to each other, either on the same line or the next line, irrespective of the total length of the clone.

3.4 How We Put Commit Sizes in Perspective

To calculate and visualize how each micro-clone inducing commit relates to the remaining distribution of commit sizes, we first calculate the churn for each commit in the repository. We do this by instrumenting `git log` to build a sequenced graph of all commits (excluding merges) in the repository, extracting the number of added and deleted lines in each commit, summing them up as the modified lines and outputting this churn integer for each commit. We then compare the churn of the micro-clone inducing commits to the overall distribution, and specifically to its median. Although our sample size of ten is too small for statistical testing, this way, we can make substantiated statements about a possible size difference between commits. We use the median (and not the average mean, for example)

Table 2 Descriptive statistics of study results

	... with all findings	... with faulty micro-clones	... with last line/stmt. bug (rel. to all, rel. to faulty)
Analysis time	June 2011 to July 2015		
Analysis software	PVS-Studio versions 4.00 to 5.27		
# of analyses	162	12 (7 %)	12 (7 %, 100 %)
# of projects	219	106 (49 %)	97 (45 %, 92 %)
# of warnings	1,891	272 (14 %)	228 (12 %, 84 %)
# of unique clones	–	263 (–)	228 (–, 87 %)

as our distributions are non-normal, it is a single real value and we compare other, singular observations to it.

4 Results

In this section, we deepen our understanding of faulty micro-clones by example and statistical evaluation.

4.1 General Description of Results

Table 2 presents basic descriptive statistics of our results for C_2. We ran the complete suite of all PVS-Studio analyses on our 219 OSS from mid-2011 to July 2015. Andrey Karpov, a software consultant by profession, gradually analyzed the different systems throughout this period, using the latest then-available version of PVS-Studio. He sorted-out false positives, so that 1,891 potentially interesting warnings with 162 different error codes remained. We then manually investigated all 1,891 warnings and found that 272 warnings with twelve distinct error codes were related to micro-cloning. Nine micro-clones contained two such warnings, so that we ended up with 263 micro-clones. The statistics at the project level reveal that our analyses could identify faulty micro-clones in half of the investigated projects. Almost all of these (92 %) contained at least one instance of the last line or statement effect.

Table 3 presents a high-level result summary of locating errors in 263 micro-clones. In total, we see that 74 % of multi-line micro-clones have a last line error and 90 % of one-liner micro-clones in the last statement.

Table 3 Summarized results

	Multi-line micro-clone	One-line micro-clone
#errors in last line/stmt.	*117 (74 %)*	*95 (90 %)*
#errors not in last line/stmt.	41 (26 %)	10 (10 %)
effect size (odds ratio)	2.9	9.5
Σ	158 (100 %)	105 (100 %)
$\Sigma\Sigma$	263	

4.2 In-Depth Investigation of Findings

To convey a better intuitive understanding of the analyses with which we identify faulty micro-clones, in the following, we select some of the 263 PVS-Studio-generated micro-clone warnings as representative examples for the most frequently occurring error codes from Table 1.

4.2.1 V501 – Identical Sub-Expressions

As Table 1 shows, the majority of micro-clone warnings are of type V501. A prime example for a V501-type warning comes from Chromium:

Example 5 Chromium

```
return !profile.GetFieldText(AutofillType(NAME_FIRST)).empty()  ||
       !profile.GetFieldText(AutofillType(NAME_MIDDLE)).empty() ||
       !profile.GetFieldText(AutofillType(NAME_MIDDLE)).empty();
```

In this one-liner micro-clone the second and third cloned statement are lexicographically identical but connected with the logical OR-operator (||), thus representing a tautology. Instead, the Boolean expression misses to take into account the surname (`NAME_LAST`), an example of the last statement effect in this tricolon.

4.2.2 V517 – Identical if-Conditions

Error code V517 pertains to having identical entry-conditions for two branches of `if`-statements.

Example 6 linux-3.18.1

```
1  if (slot == 0)
2  {
3      ...
4  }
5  else if (slot == 1)
6  {
7      ...
8  }
9  else if (slot == 0)
10 {
11     ...
12 }
```

The body of the `else if` condition following the third micro-clone on line 9 is dead code, as it can never be reached. If `slot` was indeed zero, it would already enter the first `if` condition's body.

4.2.3 V519 – Identical Assignment to Variable

Setting the value of a variable twice in succession is typically either unnecessary (and therefore a maintenance problem because it makes the code harder to understand as the first assignment is not effective), or outright erroneous because the right-hand side of the assignment should have been different. In the following V519 example from MTASA, `m_ucRed` is assigned twice, but the developers forgot to set `m_ucBlue`.

Example 7 MTASA

```
            m_ucRed = ucRed; m_ucGreen = ucGreen; m_ucRed = ucRed;
```

The detection of V519-style warnings works well for most “regular” software, but is to be taken with caution when analyzing firmware or other hardware-near code, as Example 8 demonstrates:

Example 8 linux-3.18.10

```
1  f->fmt.vbi.samples_per_line = 1600;
2  f->fmt.vbi.samples_per_line = 1440;
```

The second line sets the value of the variable `f->fmt.vbi.samples_per_line` again, even though it has just been set in line 1. Since no other method calls have been made in the further control flow of this method, the assignment in line 1 seems to have no effect. However, as the assignment is active for at least one CPU cycle, there might be threads that read its value in the meantime (for example, watchdogs on the buffer state) or there might be other intended side-effects. To be on the conservative side, we compiled the code with release settings and if the compiler optimized the first assignment away, we were sure it was indeed an error.

4.2.4 V523 – Equivalent Behavior Despite Branching

When we find a micro-clone for different branches of `if`-conditions, these could be simplified by collapsing them into one block, for example in Haiku:

Example 9 Haiku

```
1   if (flags & ATTR_COMPRESSION_MASK) {
2     hdr_size = 72;
3     /* FIXME: This compression stuff is all wrong. .... */
4     /* now. (AIA) */
5     if (val_len)
6       mpa_size = 0; /* get_size_for_compressed_....; */
7     else
8       mpa_size = 0;
9     ...
10  }
```

It is, however, more likely that `mpa_size` should have been set to a different value in the `else`-branch. The code context of this micro-clone seems highly suspicious, as it mentions in line 3 that “[t]his compression stuff is all wrong,” and the detected erroneous micro-clone fits to this comment.

4.2.5 V524 – Equivalent Function Bodies

Two cloned functions with the same content are highly suspicious. In our Example 10, line 5 should call `PerPtrBottomUp.clear()`. This also serves as one rare example of a two-instance micro-clone where the origin succeeds the target ($\delta_{E_{10}} = -1$).

Example 10 clang

```
MapTy PerPtrTopDown;
MapTy PerPtrBottomUp;

void clearBottomUpPointers() {
  PerPtrTopDown.clear();
}

void clearTopDownPointers() {
  PerPtrTopDown.clear();
}
```

4.2.6 V537 – Suspicious Use of Variable or Statement

An illustrative example for a V537 finding comes from Quake III, where PVS-Studio alerts us to review the use of `rectf.X`:

Example 11 Quake III

```
 rect->X = roundr(rectf.X);
 rect->Y = roundr(rectf.X);
```

The rectangle's y-coordinate is falsely assigned the rounded value of `rectf.X` in the second (i.e., last) line of this micro-clone.

4.2.7 V656 – Two Variables Bear Identical Value

V656 checks for different variables that have the same initializing function. As a result, we need to check warnings of type V656 carefully, as they bear a high potential for false-positives. One example for a false-positive is that the two variables are supposed to start with the same value, and are then treated differently in the downstream control-flow. All V656-related micro-clones in our sample stem from LibreOffice.

Example 12 LibreOffice

```
 maSelection.Min() = aSelection.Min();
 maSelection.Max() = aSelection.Min();
```

Here, `maSelection.Max()` is assigned not the maximum value of `aSelection`, but its minimum, clearly representing an error.

4.2.8 Counterexample

As we have already seen in Example 12, not for all instances of an erroneous micro-clone does the problem lie in the last line or statement. Take this rare counterexample from Chromium, which we counted towards the 12 instances of an error in line 2 of a three-liner micro-clone (see Table 4):

Example 13 Chromium

```
 if (std::abs(data_[M01] - data_[M10]) > epsilon ||
     std::abs(data_[M02] - data_[M02]) > epsilon ||
     std::abs(data_[M12] - data_[M21]) > epsilon)
```

Table 4 Error distribution for micro-clones with ⩾ 2 lines

#errors in line	#total lines 1	2	3	4	5	6	7	8	9	>9
1		8	0	0	1	0	0	0	0	0
2		*66*	12	3	1	0	0	0	0	0
3			*22*	4	0	1	1	0	0	0
4				*15*	0	0	0	0	0	0
5					*6*	1	0	1	0	1
6						*3*	0	0	0	1
7							*1*	0	0	1
8								*0*	1	0
9									*2*	4
>9										*2*
Σ	0	74	34	22	8	5	2	1	3	9
$\Sigma\Sigma$					158					
p		10^{-106}	10^{-27}	10^{-15}	10^{-5}	0.0487	0.534	0.437	0.135	

In line 2, the engineers deducted `data_[M02]` from itself. However, they meant to write:

```
2       std::abs(data_[M02] - data_[M20]) > epsilon ||
```

4.3 Statistical Evaluation

Table 4 shows the error-per-line distribution of our 158 micro-clones consisting of several lines, and Table 5 that of our 105 micro-clones within one single line. Cells with gray background are non-sensible. For example, in a micro-clone of 2 lines length, no error can occur in line 3. The yellow diagonal highlights errors in the last line or statement.

For each column in Tables 4 and 5, we performed a Pearson's χ^2 test on a $p = 0.05$ significance level to see whether the individual distributions are non-uniform. The resulting p-values, reported in the last row, are only meaningful for micro-clone lengths with enough empirical observations, which are columns 2-6 in Table 4 and columns 2-4 in Table 5.

For RQ1 and RQ2, we got significant p-values for micro-clones consisting of 2, 3, 4, 5 or 6 lines and for micro-clones consisting of 2, 3, or 4 statements ($p < 0.05$). This means that we can reject the null hypothesis that errors are uniformly distributed across statements or lines. Instead, the distribution is significantly skewed towards the last line or statement.

Table 5 Error Distribution for Micro-Clones within One Line

#errors in statement	#total statements 1	2	3	4	5	> 5
1		1	0	0	0	0
2		*71*	4	2	0	0
3			*18*	1	0	0
4				*7*	0	0
5					*0*	0
>5						*1*
Σ	0	72	22	10	0	1
$\Sigma\Sigma$			105			
p		10^{-73}	10^{-13}	10^{-4}		

We would expect similar findings for longer micro-clones, but these were too rare to derive statistically valid information, shown by gray areas of the last row in Tables 4 and 5.

We can summarize the results across micro-clone lengths into the two events “error not in last line or statement” and “error in last line or statement”, shown in Table 3. Our absolute counts show that in micro-clones similar to Example 1, the last line is almost thrice as likely to contain a fault than all previous lines taken together. When looking at the individual line lengths in Table 4, the last line effect is even as high as a nine-fold increased error-proneness for the oft-appearing clone lengths 2, 4 and 5. The results for cloned statements in micro-clones within one line, like Example 2, are stronger still: We found the last statement to be 9.5 times more faulty than all other statements taken together. In fact, for the 72 micro-clones consisting of two statements, the last statement was the faulty one in all but one case.

In total, our findings confirm the presence of a pronounced last line and last statement effect, accepting both RQ 1 and RQ 2.

4.4 Origin of Micro-Clones

Having found a large number of seemingly trivial micro-clone-related bugs in OSS projects, we were curious about the reasons for its presence. In RQ 3, we therefore ask:

RQ 3 What are the reasons for the existence of faulty micro-clones and the last line effect in particular?

In this section, we first analyze the origin of micro-clone instances, and examine which technical and psychological reasons might play a role for the existence of micro-clones.

Table 6 shows the results of the copy origin analysis broken down per clone length. For it, we disregarded micro-clones for which we could not agree on the order of their clones, leaving us with 245 out of 263 clone pairs.

In Fig. 2, we plot the distribution of the copy origin. The figure shows that for 165 out of 245 micro-clones (67 %), the first clone instance of a micro-clone is the influential one, with a large drop toward the second (18 %) and subsequent gradual drops from the second to the third (9 %) and fourth (3 %). Only in the remaining 4 % of cases does the influencing clone instance lie beyond the fourth line or statement in a micro-clone. This seems to indicate that the first line is most influential for the outcome of a clone. However, our distribution of micro-clones itself is highly skewed toward 2-liner micro-clones. It follows naturally that in

Table 6 Clone Length (Horizontal) and Likely Clone Origin (Vertical)

clone length / origin	1	2	3	4	5	6	7	8	9	>9
1	0	132	22	7	2	1	0	0	1	0
2		3	28	3	4	1	1	1	0	0
3			1	20	0	0	0	0	1	1
4				0	2	1	1	0	0	3
5					0	2	0	0	0	0
6						0	0	0	0	2
7							0	0	1	3
8								0	0	1
9									0	0
>9										0
Σ	0	135	51	30	8	5	2	1	3	10
$\Sigma\Sigma$					245					

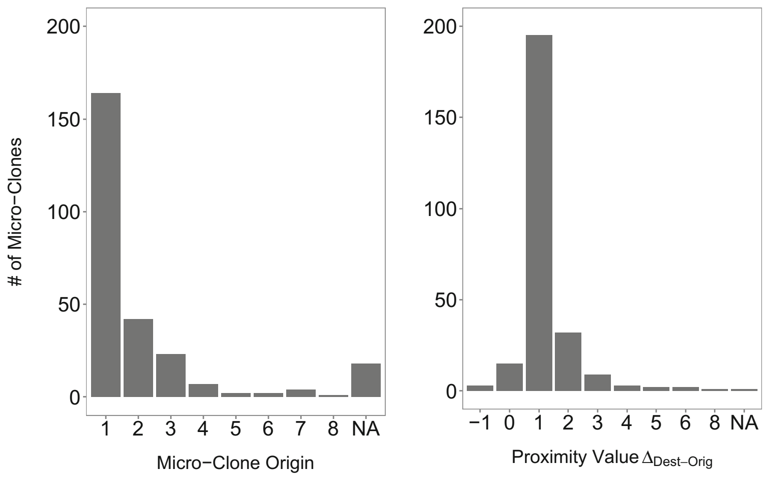


Fig. 2 Copy Origin Distribution (*left*) and Proximity Distribution of Copy Origin and Destination within Micro-Clones (*right*)

most instances of a two-liner micro-clone, the origin lies in the first line. When considering the 117 micro-clones which are longer than two clone instances in Table 6, we find that the copy origin is the first line only for 33 micro-clones (28 %). As the average length for these 117 micro-clones is 4.9, we would expect 20 % of copy origins to be in the first line, even for a uniform distribution. Our 28 % indicate that the first line only exhibits a mild influence when correcting for the influence of 2-liner micro-clones. However, In 2-liner micro-clones, the first line is almost always the origin (rather than the second line influencing the first).

Figure 2 plots the distribution $\Delta_{Dest-Orig}$ (see Section 3.3). It shows that over 84 % of clone instances appear in the immediate mutual neighborhood of each other (220 out of 245), i.e. $\Delta_{Dest-Orig} \leqslant 1$. In 89 % of these cases (195 out of 220), $\Delta_{Dest-Orig} = 1$ or $\Delta_{Dest-Orig} = -1$, which means that the erroneous instance succeeds the correct instance in either the next line or statement. Preceding it, i.e. $\Delta_{Dest-Orig} = -1$, is much rarer (3 out of 220). When we disregard 2-liner or 2-statement clones, which naturally appear next to each other, we obtain that 81 % of clone pairs appear in mutual neighborhood (66 out of 81). We can therefore summarize these findings with two general observations:

1. For 2-liner micro-clones, the first line is almost always the influencing origin. When correcting for the effects of short 2-liner code clones, in general, the first line of a micro-clone seems to only exhibit a mild additional influence as a source.
2. Instead, the influencing and the erroneous clone instance appear in direct textual and visual neighborhood in the source code in four out of five micro-clones. Moreover, in nine out of ten of these cases, the erroneous clone instance appears after its influencing origin.

4.5 Developer Interviews

In C_2, we approached ten authors of real-world committed micro-clones with excerpts of the micro-clone they authored and additional contextual information. We then asked them whether they remembered

1. how they mechanically created the micro-clone (e.g., by copy and pasting).
2. how the particular error referenced occurred or slipped-through.
3. which situation they were in when they created the commit, supplied with the local time and date of the commit.
4. in which stages of development and how often similar micro-clones are generally created in their experience.

Table 7 gives an overview of the ten micro-clones and associated seven interviews which we lead asynchronously via email and Skype. The table denotes the sampled projects and commits, the creation date of the commits, their median and individual sizes in terms of churn, and the total number of commits in the project. To protect the identity of interviewees *I1*-*I7*, we do not connect the IDs with commits in the table and also anonymize all subsequent code fragments. If we received no reply after one week, we sent a one-time reminder to participate to the interviewee. In the following, we summarize the insights we obtained from the interviews. We discuss interviews *I1*, *I2*, *I4*, *I6*, and *I7* at length. As we reached a preliminary saturation, our abbreviated findings here summarize the other remaining interviews.

One interviewee replied that he has "no interest." Another interview ended because the participant replied that the commit was too long ago and he does not remember it. In one instance, `7b37fbb`, the interviewee *I1* told us that he merely refactored and did not author this piece of code originally (hence we report six interviews with authors in Table 7). He forwarded us to the real author of the code, whom we also interviewed (`6b7fcb4`).

We asked *I2* about the micro-clone:

Example 14 Anonymized I1

```
if (!has_mic && !has_mic) {
```

He told us that the mistake was not a copy-and-paste mistake. Rather, he typed `!has_mic` when he should have typed `!has_audio` instead. In his experience, this happens a lot when working with code in which one types the same words repetitively. He observed that "I was not under any major stress at the time," but that "I will note that when working with very large changes it is much easier for something like this to be missed." He

Table 7 Descriptive Statistics of Developer Interviews and Commit Size Analysis of sampled repositories before 6.10.2016

Project	Sampled Commit	Local Commit Date	Commit Churn	Median Churn	#Commits	Replies
Chromium	`2db5310`	2010-09-30 20:53	123	43	639,564	4/4
	`6b7fcb4`	2011-02-23 05:57	1,220			
	(`7b37fbb`)	(2011-03-07 16:16)	(1,635)			
	`47fcb0e`	2012-10-24 3:52	1,627			
LibreOffice	`b90bc10`	2008-08-19 22:06	103,083	18	438,994	0/2
	`44cfc7c` (rebase)	2012-10-09 12:22	470			
Samba	`781ed1f`	2005-12-09 05:21	45	16	241,276	1/1
Mesa 3D	`0ff3b2b`	2010-07-26 23:56	108	21	99,115	1/2
	`45124e0`	2010-12-07 21:37	251			
libjingle	`562554d`	2010-09-30 20:53	110,184	212	341	1/1
$\sum$	10					7/10 (6 authors)

added that the real error was not having a unit test that covers this line and that the reviewer missed the absurdity of the pattern `!a && !a`, too.

I4 answered that, while he did not remember this case specifically, he reconstructed what likely happened for the micro-clone of the form:

Example 15 Anonymized I4

```
  return
  field.type == trans("string") ||
  field.type == trans("twitter") ||
  field.type == trans("mail") ||
  field.type == trans("http") ||
  field.type == trans("email") ||
  field.type == trans("string");
```

When creating such micro-clones, he usually comes up with the first clone instance `field.type == trans("string") ||` and copy-and-pastes it several times, ending up in a sequence like:

Example 16 Anonymized I4

```
field.type == trans("text") ||
field.type == trans("text") ||
...
```

He reported that he does "not carefully count how many repetitions there are – I just guesstimate." As a last step, he would also remember to delete any extraneous lines, but he assumes that he did not remember in this case or got distracted. During the origin analysis (see Section 3.3), we also found that two refactorings on this micro-clone were performed, but the original error stayed. This happened because developers relied on a tool to do the transformation for them and did not read the code carefully. *I4* concluded that he often uses these mechanics for creating micro-clones, "but I usually remember to pare down any extraneous lines." Similar to *I1*, he also stated that it should be caught by either code review or testing.

I6 answered that "it has been a while, but [...] this seems like [a] copy/paste bug to me. Not uncommon." He also said "I see (and do) this kind of thing all the time." To move fast and save typing, *I6* created the micro-clone by copy-and-pasting, then modifying each line by varying it. "The last line got missed." His explanation is that he forgot to modify the last micro-clone instance, since "usually, my mind has moved on to less mechanical thought. But then the mechanical actions gets botched." While *I6* did not recall the day particularly, they are "always trying to move fast to get improvements out." He also said that he sees micro-clones "all the time," at least 10 times per day. "Of those 10, perhaps 9 get caught in self review or by the compiler. The last one gets caught by other reviewers or unit tests mostly. But on occasion, say once a month [...], this kind of [bug] makes it into shipping code that affects end users."

I7 authored a micro-clone of the format

Example 17 Anonymized I7

```
else if(depth > 0 && width > 0 && width > 0)
```

He remembered that he "just typed it out, no copy/paste" and missed it because "I was probably in a hurry and was not focused." While he could not remember the specific date, he noted that he is "always pretty busy in general."

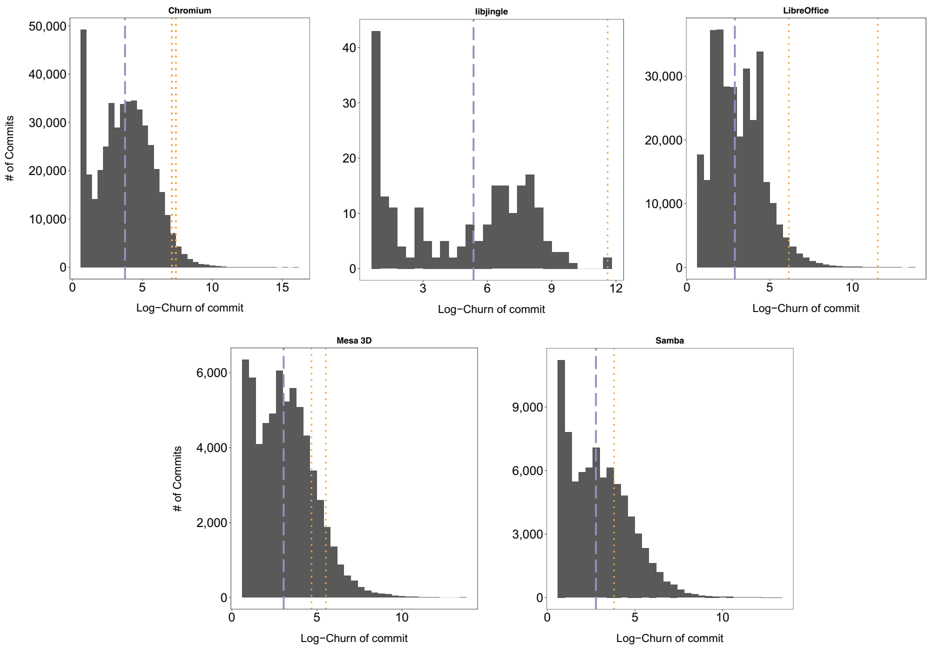


Fig. 3 Median commit size over whole repository history (*dashed blue*) and commit size (as logarithmic churn) of individual micro-clone introducing commits (*dotted orange*)

From the interviews, it seemed that one factor that might affect the likelihood of faulty micro-clones to pass through the various measures of defense the interviewees mentioned, might be the size of the commit. If this is the case, then micro-clone inducing commits should be abnormally large. The term "abnormally large" only makes sense in the context of each project's relative commit sizes. In Fig. 3, we therefore compare the size of the sampled micro-clone inducing commits to the median commit size in each project. The figures show that all micro-clone inducing commits were orders of magnitude larger than the median commit sizes in each project.

4.6 Usefulness of Results

Having unveiled a large number of potential bugs in OSS, we wanted to help the OSS community and see if our findings represented bugs that would be worth fixing in reality. We approached the OSS projects by creating issues with our findings in their bug trackers. Many of our bug reports lead to quality improvements in the projects, like fixing the validation bug from Example 2 in Chromium.[9] The search query `pvs-studio bug | issue`[10] shows numerous bug fixes in Firefox, libxml, MySQL, Clang, samba and many other projects

[9] https://codereview.chromium.org/7031055

[10] www.google.com/search?q=pvs-studio+bug+|+issue

based on our findings. As one such example, on October 11th 2016 in commit `caff670`, we fixed a micro-clone-related issue that had existed in samba since 2005.[11]

5 Discussion

In this section, we discuss our results by merging the observed bug patterns with our psycho-cognitive analysis. We end with an explanation of possible threats to the validity of our conclusions.

5.1 Technical Complexity & Reasons

Technical reasons that could play a role for the existence of the last line effect would assume that the last line in a micro-clone is technically more complex in comparison to the other lines, and thus more likely to contain an error. This would include that the last line is for example not checked by the compiler, or that, when an Integrated Development Environment (IDE) is used and the last line indeed written as the last action in this editor window, perhaps the compiler would not react fast enough to check it. This is not true for two reasons:

1. Modern IDEs typically do not lag behind in syntax checking.
2. The last line or statement micro-clone instances are grammatical, i.e. a compiler error that would draw attention to them does not occur.

On the other hand, if IDEs and compilers did include checks for micro-clones, they could help developers catch them before committing.[12]

Another technical reason might be that coming up with the last statement in a series of statements might be harder than the ones before. However, when observing any of the Examples 1, 2, 5, 7 and 11, it becomes clear that the opposite is the case: Because all clone instances in a micro-clone follow the same pattern, the hardest to come up with, if any, is the first instance. The succeeding instances simply replicate its pattern.

5.2 Psychological Mechanisms & Reasons

As technical reasons are not a likely cause for the last line effect, we consider here psychological mechanisms that might underlie this effect. We turned to a professor in cognitive psychology (the fourth author of this paper) and presented our findings to him. In the following, we give an initial overview of possible psychological effects. These psychological reasons are preliminary at this point, because a more detailed analysis would require psychological experimentation in which the actual process of producing these errors is observed, rather than reconstructed by an origin analysis (see Section 3.3) and remembered in interviews (see Section 4.5).

In cognitive psychology, action slips are errors that occur during routine tasks and have been widely studied (Anderson 1990). A typical example would be to put milk in a coffee twice instead of milk and sugar. Our analysis on the origin of micro-clones concluded that developers follow a wide variety of different mechanical patterns to create micro-clones.

[11] https://bugzilla.samba.org/show_bug.cgi?id=12373

[12] Clang started to implement our analyses, see https://llvm.org/bugs/show_bug.cgi?id=9952.

One of these patterns is "[write first clone instance], [copy], [copy], ..., [modify], [modify], ...", see *I4, I6*. Our interviews and origin-analysis also show that developers equally follow the pattern "[write first clone instance], [copy, modify], [copy, modify], ..." In some extreme cases micro-cloning in our data set, this action sequence must have been repeated 34 times. Even though they use different mechanical methods, the task software developers are performing in producing micro-clones can always be seen as a sequential action task with different levels of automation and manual effort. From a psycho-cognitive viewpoint, errors developers introduce while producing micro-clones are thus characterized as typical action slips.

While differing on the details, models for sequential action control assume that noise is the main explanation for action slips (Botvinick and Plaut 2004; Cooper and Shallice 2006; Trafton et al. 2011). By *noise*, we refer to any task-irrelevant mental representations, which includes external stress such as deadlines and internal factors such as large commits, that might draw the developer's attention. Sequential action control models provide a useful theoretical framework for speculating about the psychological mechanisms behind the last line effect. At this point, we only know the faulty micro-clones instances, and their location, but we have no detailed process information on how they came to be. However, as Section 4.4 showed, the anecdotal evidence from interviews as well as our technical origin analysis does allow us to make informed inferences about the creating of an erroneous micro-clone instance. The basic operations that the programmer performs are: copying and editing. Consider Example 1 again. The editing step here involves two sub-steps, updating the variable and updating the value.

Example 1 TrinityCore

```
1  x += other.x;
2  y += other.y;
3  z += other.y;
```

Here, line 3 contains an error. It appears that line 2 was copied to produce line 3. The first update was performed correctly (change `y` into `z`) but the second editing sub-step was not performed, thus producing the error. In principle, line 1 could have been copied twice with the editing steps having been performed on the two lines. However, the presence of a y rather than x in line 3 suggests that line 2 was copied. Section 4.4 shows that in most cases of micro-clones with more than two lines, the previous line was copied. This suggests that in such micro-clones, the sequence of actions was as follows: "[copy, modify, modify], [copy, modify, modify], ..."

Models of action control assume that action slips occur because of noise. Such noise is more likely to occur near the end of a sequence because the programmer's focus might prematurely shift to the next task, for example subsequent lines of code that need to be produced (see evidence from *I6*). As noted earlier, there are subtly different psychological explanations for why such noise might occur. To take just one account (Cooper and Shallice 2006), the last line effect might occur because the wrong action schema is selected (e.g., the engineer is already mentally working on the next lines rather than completing the current one).

Although none of the engineers noted to have experienced extraordinary stress levels at the time of the creation of the clone, the statements from *I6* and *I7* stand out, who indicated a general sense of business and desire to move fast. When considering the local commit dates of when erroneous micro-clones in Table 7, it stands out that only two were created during core office hours, even though many interviewees did this as part of their job. Tiredness is known to reduce brain efficiency and affect the working memory (Kane et al. 2007). This could indicate that tiredness and a general time pressure might play a critical role in the creation of erroneous micro-clones.

More than time pressure, however, we found that all micro-clone inducing commits (and even refactorings) were exceptionally large – orders of magnitude larger than a normal commit in the repositories. We therefore purport that commit size is an important, perhaps the dominant noise factor, that makes these errors go unnoticed. This finding corroborates well with the explanations of a working memory overload and *I1's* observation that the resulting amount of code is very hard to oversee.

Our interviews with developers indicated that creating short-lived micro-clones might be common in software development, but that the developers usually catch them early, or at least during their own or someone else's review of the code (Beller et al. 2014). The cognitive error in the remaining micro-clones we observed in this study is thus not only a production error, but also a proofreading error (Healy 1980): During revision of the code, the engineer fails to notice the error in the last and other lines. In fact, our interviews suggest that this seemed to happen twice for the micro-clones that made to production: once, during self-review and then at least one second time during code review by a peer. One plausible reason why this proofreading error is more likely to occur in the last line than in earlier ones could be because it is an action slip. The mind is already focused on the next task (e.g., implementing a new feature) before the current one (proofreading), has been completed. Yet another account could be that the error is less detectable because several very similar statements in a row have to be proofread. This could cause the reading of the final statement to be faster and therefore more superficial. Moreover, the visual closeness of origin and target in micro-clones might simply make it more difficult to differentiate between the individual lines. Research on proofreading suggests that familiarity (operationalized as word frequency) leads to shorter processing times and has a negative impact on the ability to detect spelling errors in text (Moravcsik and Healy 1995).

All potential causes suggest that developers are more likely to conduct last-line-type errors in situations when their attention span is reduced through noise. Possible causes for noise with a negative impact on micro-cloning in particular seem to be large commit sizes, and possibly high workload, stress, being distracted, and tiredness (O'Malley and Gallas 1977). Conversely, our results also suggest that developers' ability to control irrelevant noise from the environment (Fukuda and Vogel 2009), i.e. their ability to focus attention, plays an important role in how likely a micro-clone is going to be created with an action slip related error.

5.3 Threats to Validity

In this section, we show internal and external threats to the validity of our results, and how we mitigated them.

5.3.1 Internal Threats

An important internal threat to this study concerns how to determine in which line the error lies. Given Example 2, any of the two statements could be counted as the one containing the duplication. However, reading and writing source code typically happens from top to bottom and from left to right (Siegmund et al. 2014). Therefore, the only natural assessment is to flag lines and statements as problematic according to this strict left-right and top-down visual reading order: In Example 2, only when we have read the second statement do we know it is a duplicate of the first. We hence flag the second statement as the one containing the error. Moreover, in many cases, as in Example 2, the program context around the micro-clone (here the definition order of the variables `host` first and then `port_str`) imposes a natural logical order for the remainder of the program (first check `host`, then `port_str` in line 3). In order to reduce personal bias, we also separated the list of findings to triage across the first two authors, and then discussed unclear cases. If we could not reach agreement, we discarded said finding. In this process, we also re-classified all original previous 202 findings (Beller et al. 2015) and found almost total agreement with our previous assessment. Since flagging erroneous lines is a well-defined task under these circumstances, we are sure there is a high inter-rater reliability, ensuring the repeatability of our study.

It is likely that our checkers are not exhaustive in detecting all faulty micro-clones. This poses only a small threat to the validity of our results, as we do not claim to cover all micro-clones. We believe to have captured a major part of the micro-clone population through extending our checkers to 12 (see Table 1). Evidence that our analyses capture a major source of bugs comes from the fact that only our core checkers V501, V517, V519 and V537 add a substantial share of the results and that van Tonder and Le Goues found more than 24,000 faulty micro-clones using a subset of our checkers (van Tonder and Le Goues 2016).

While we are confident about the results of our origin-destination analysis, we do not know how the clones were created and modified by the software developer. Our a-posteriori repository mining approach assumes a top-to-bottom reading order of blocks and a left-to-right reading order for individual lines. We know that developers "jump" in the code when reading a file, only focusing on what seems important to solve the task at hand (Busjahn et al. 2015; Siegmund et al. 2014). However, in order to understand small coherent logical units, such as micro-clones, developers must necessarily read in the control-flow-direction of the software – which is top-to-bottom, left-to-right. In particular, it would be interesting to see 1) how many times the copy-paste-pattern "`ctrl+c`, `ctrl+v`" was used, 2) in which order micro-clones are created, and 3) in which order micro-clones are read and changed, if developers need to modify them during maintenance. In order to get to know such information, we would require to study how developers work in-vivo, similar to the WatchDog plugin (Beller et al. 2015, 2015, 2016). To that end, we could reuse parts of CloneBoard, which captures all cut, copy and paste actions in Eclipse (de Wit et al. 2009).

Given these limitations, our psychological analysis is partly speculative at this point. A more detailed analysis requires psychological experimentation in which the process of producing these errors is examined in-vivo. With our choice of research methods, we might potentially miss subtle steps in the creation of micro-clones. However, we believe that it is very difficult to expose faulty micro-cloning in a laboratory setting, as our interviews indicate that it requires a long time to expose a relatively small number of micro-clones. Moreover, due to the artificial nature of the experiment, a possible time limit and the fact that participants typically over-perform in experiments (Adair 1984), they might not create faulty micro-clones at all. Since the results of our mixed-methods case studies corroborate each other, we believe to have acquired an accurate set of initial reasons for the existence of faulty micro-clones and the last line effect in particular.

5.3.2 External Threats

An external factor that threatens the generalizability is that PVS-Studio is specific to C and C++. C is one of the most commonly used languages (Meyerovich and Rabkin 2013). Therefore, even if our results were not generalizable, they would at least be valuable to the large C and C++ communities. However, our findings typically contain language features common to most programming languages, like the variable assignments, if clauses, Boolean expressions and array uses in Examples 1, 2, 5, 7 and 11. Almost all programming languages have these constructs. Thus, we expect to see analogous results in at least C-inspired languages such as Java, JavaScript, C#, PHP, Ruby, or Python. While our overall corpus of findings is large, the average number of ∼1.2 micro-clones per project is rather small (see Table 2). This could be because PVS-Studio's analyses for defective micro-clones are not exhaustive, and that the projects we studied are stable, production systems with a mature code base containing relatively little trivial errors. Our interviews gave another explanation: extensive testing and code-reviewing significantly decreases the number of faulty micro-clones that make it into production.

6 Related Work

Duplicated or similar code fragments are famously known as "code clones," yet their definition has remained somewhat vague over the last decade (Roy et al. 2014). This vagueness is reflected in the definitions "[c]lones are segments of code that are similar according to some definition of similarity" by Baxter et al. (1998) and "code clones [...] are code fragments of considerable length and significant similarity" by Basit and Jarzabek (2007). The latter definition identifies clones as long enough pieces of code that share sufficiently many traits, while the first has no such requirements. A widely-used definition categorizes clones into three classes (Koschke 2007): Type 1 clones are textually and type 2 clones syntactically (modulo identifier renamings) identical. Type 3 clones have further-reaching syntactic modifications and type 4 clones are only functionally identical (Roy et al. 2014). However, this general classification is agnostic about, for example, code clone length. Subsequently, researchers developed a plethora of more specific clone definitions (Koschke 2007; Balazinska et al. 1999; Kapser and Godfrey 2003). In this study, we add to these taxonomies the concept of very short, but closely related code clones that are located below the lower limit of "considerable length," with often no more than two duplicated statements within one clone instance. We call such extremely short duplicated pieces of code, *micro-clones*.

In the following, we compare traditional code clone detection mechanisms to how we detect micro-clones. In a 2007 comparison and evaluation of clone detection tools, Bellon et al. evaluated six clone detectors for C and Java (Bellon et al. 2007). Depending on the clone detector, clones had to be at least six lines or 25 tokens long in their experiment. In 2014, Svajlenko and Roy performed a similar study and compared the recall performance of eleven modern clone detection tools (Svajlenko and Roy 2014). In their configuration of the clone detectors, they used minimal clone lengths of 50 tokens, 15 statements, or 15 lines (Svajlenko and Roy 2014). These thresholds are too large to be able to detect micro-clones. However, traditional clone detectors need them to avoid a large number of false positives. Our approach circumvents this problem by only detecting faulty micro-clones.

In direct follow-up research on our initial investigation (Beller et al. 2015), van Tonder and Le Goues performed a large-scale search for micro-clones in 380,125 Java repositories (van Tonder and Le Goues 2016). They found 24,304 faulty micro-clones, demonstrating and solidifying our assumption that micro-clones are a wide-spread phenomenon across software projects. By providing 43 patches to fix faulty micro-clones of which 43 were promptly integrated, they show that developers value the removal of micro-clones and that it can be automated at scale.

Empirical investigations with traditional clone detectors suggest that ∼9 % to 17 % is a typical portion of clones in the code base of software systems (Zibran et al. 2011), considering all type 1, 2, and 3 clones (Koschke 2007). Outliers in the so-called "clone coverage" may be lower than 5 % (Roy and Cordy 2007) and higher than 50 % (Rieger et al. 2004; Roy et al. 2014). These ratios do not include micro-clones, which we have shown to be a frequent source of bugs in numerous OSS in this study. In the larger perspective of how prevalent code clones are in systems, micro-clones might lead to an increased perception of clones in the code, and to a much higher clone coverage, at least when considering a "micro-clone coverage" measure. A high clone coverage is generally thought to be problematic, since numerous studies have shown that it is positively correlated with bugs and inconsistencies in the system (Chatterji et al. 2011; Göde and Koschke 2011; Inoue et al. 2012; Xie et al. 2013).

7 Future Work & Conclusion

In this section, we describe possible extensions of our study and draw conclusion.

Because our study focuses on faulty micro-clones, we cannot make predictions about how many of all micro-clones are erroneous. A promising future research direction is to develop a clone detector that can reliably detect all micro-clones, and then to see how many are actually defective. This gives an indication of the scale of the problem at hand. Anecdotal evidence from interviews suggests that micro-cloning seems to happen quite often and catching it consumes precious code review and testing iterations. To catch faulty micro-clones early, including our checkers for micro-clones into the IDEs of developers seems to be fruitful direction for future work.

Our initial psychological examination of the effect warrants a larger psychological controlled experiment, that, we believe, might be associated with a high risk of not exposing enough faulty micro-clone creations. Already existing tooling could help enable this study on a technical level.

In 219 open source projects, we found 263 faulty micro-clones. Our analysis shows that there is a strong tendency for the last line, and an even stronger tendency for the last statement to be faulty, called the last line effect. In fact, the last line in a micro-clone is three times as likely to contain a fault than any of the previous lines combined, and the last statement almost ten times as likely as any of the previous statements combined.

Psychological reasons for the existence of the last line effect seem to be largely the result of *action slips*, where developers fail to carry out a repetitive and easy process correctly, caused by working memory overload by noise. We have evidence suggesting that the effect is largely caused by the way developers copy-and-paste code. Developers seem to have a psychological tendency to think changes of similar code blocks are finished earlier than they really are. This way, they miss one critical last modification. Important reasons for noise seem to be abnormally large commit sizes, and possibly tiredness and stress.

Because of this observation, we advise programmers to be extra-careful when reading, modifying, creating, or code-reviewing the last line and statement of a micro-clone, especially when they copy-and-paste it. Moreover, our finding that faulty micro-clones were only present in abnormally large commits emphasizes the importance of small, manageable commits. This knowledge can help developers alleviate bugs due to faulty micro-clones, both while writing and reviewing code. Developers can spot mental situations in which they are likely to commit errors due to an overload of their working memory, and pay attention to avoid them. With PVS-Studio, we have developed an automated tool that supports developers to spot when such errors have “slipped through” pre-release, for example during code review.

Acknowledgments We thank Diomidis Spinellis for an inspiring conversation during ICSE’15 in the “Mercato Centrale.” For reviewing drafts of this paper, we thank Maurício Aniche, Joseph Hejderup, and Mozhan Soltani.

References

Adair JG (1984) The Hawthorne effect: a reconsideration of the methodological artifact. J Appl Psychol 69(2):334–345

Anderson JR (1990) Cognitive psychology and its implications. WH Freeman/Times Books/Henry Holt & Co

Balazinska M, Merlo E, Dagenais M, Lagüe B, Kontogiannis K (1999) Measuring clone based reengineering opportunities. In: Proceedings of the international software metrics symposium (METRICS). IEEE, pp 292–303

Basit HA, Jarzabek S (2007) Efficient token based clone detection with flexible tokenization. In: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT international symposium on foundations of software engineering (ESEC/FSE). ACM, pp 513–516

Baxter ID, Yahin A, de Moura LM, Sant’Anna M, Bier L (1998) Clone detection using abstract syntax trees. In: Proceedings of the international conference on software maintenance (ICSM). IEEE, pp 368–377

Beller M, Bacchelli A, Zaidman A, Juergens E (2014) Modern code reviews in open-source projects: Which problems do they fix? In: Proceedings of the 11th working conference on mining software repositories. ACM, pp 202–211

Beller M, Bholanath R, McIntosh S, Zaidman A (2016) Analyzing the state of static analysis: a large-scale evaluation in open source software. In: Proceedings of the 23rd IEEE international conference on software analysis, evolution, and reengineering. IEEE, pp 470–481

Beller M, Gousios G, Panichella A, Zaidman A (2015) When, how, and why developers (do not) test in their IDEs. In: Proceedings of the 10th joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE). ACM

Beller M, Gousios G, Zaidman A (2015) How (much) do developers test? In: 37th International conference on software engineering (ICSE). ACM, pp 559–562

Beller M, Levaja I, Panichella A, Gousios G, Zaidman A (2016) How to catch ’em all: watchdog, a family of ide plug-ins to assess testing. In: 3rd International workshop on software engineering research and industrial practice (SER&IP 2016). IEEE, pp 53–56

Beller M, Zaidman A, Karpov A (2015) The last line effect. In: 23rd International conference on program comprehension (ICPC). ACM, pp 240–243

Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E (2007) Comparison and evaluation of clone detection tools. IEEE Trans Softw Eng 33(9):577–591

Bland JM, Altman DG (2000) The odds ratio. Bmj 320(7247):1468

Botvinick M, Plaut DC (2004) Doing without schema hierarchies: a recurrent connectionist approach to routine sequential action and its pathologies 111:395–429

Busjahn T, Bednarik R, Begel A, Crosby M, Paterson JH, Schulte C, Sharif B, Tamm S (2015) Eye movements in code reading: relaxing the linear order. In: Proceedings of the international conference on program comprehension (ICPC). ACM, pp 255–265

Chatterji D, Carver JC, Massengil B, Oslin J, Kraft N et al (2011) Measuring the efficacy of code clone information in a bug localization task: an empirical study. In: Proceedings of the international symposium on empirical software engineering and measurement (ESEM). IEEE, pp 20–29

Cooper R, Shallice T (2006) Hierarchical schemas and goals in the control of sequential behaviour, vol 113

de Wit M, Zaidman A, van Deursen A (2009) Managing code clones using dynamic change tracking and resolution. In: Proceedings of the international conference on software maintenance (ICSM). IEEE, pp 169–178

Fukuda K, Vogel EK (2009) Human variation in overriding attentional capture. J Neurosci 29(27):8726–8733

Göde N, Koschke R (2011) Frequency and risks of changes to clones. In: Proceedings of the international conference on software engineering (ICSE). ACM, pp 311–320

Healy AF (1980) Proofreading errors on the word the: new evidence on reading units. J Exper Psychol Human Percep Perform 6(1):45

Inoue K, Higo Y, Yoshida N, Choi E, Kusumoto S, Kim K, Park W, Lee E (2012) Experience of finding inconsistently-changed bugs in code clones of mobile software. In: Proceedings of the international workshop on software clones (IWSC). IEEE, pp 94–95

Juergens E, Deissenboeck F, Hummel B, Wagner S (2009) Do code clones matter? In: Proceedings of the international conference on software engineering (ICSE). IEEE, pp 485–495

Kane MJ, Brown LH, McVay JC, Silvia PJ, Myin-Germeys I, Kwapil TR (2007) For whom the mind wanders, and when an experience-sampling study of working memory and executive control in daily life. Psychol Sci 18(7):614–621

Kapser C, Godfrey M (2003) A taxonomy of clones in source code: the re-engineers most wanted list. In: 2nd International workshop on detection of software clones (IWDSC-03), vol 13

Kapser CJ, Godfrey MW (2008) Cloning considered harmful–considered harmful: patterns of cloning in software. Emp Softw Eng 13(6):645–692

Kim M, Bergman L, Lau T, Notkin D (2004) An ethnographic study of copy and paste programming practices in oopl. In: Proc. International symposium on empirical software engineering (ISESE). IEEE, pp 83–92

Koschke R (2007) Survey of research on software clones. In: Koschke R, Merlo E, Walenstein A (eds) Duplication, redundancy, and similarity in software, no. 06301 in Dagstuhl seminar proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI). Schloss Dagstuhl, Dagstuhl. https://web.archive.org/web/20161024110147/http://drops.dagstuhl.de/opus/volltexte/2007/962/

Meyerovich L, Rabkin A (2013) Empirical analysis of programming language adoption. In: ACM SIGPLAN notices, vol 48. ACM, pp 1–18

Moravcsik JE, Healy AF (1995) Effect of meaning on letter detection. J Exper Psychol Learn Memory Cogn 21(1):82

O'Malley JJ, Gallas J (1977) Noise and attention span. Percep Motor Skills 44(3):919–922

Rieger M, Ducasse S, Lanza M (2004) Insights into system-wide code duplication. In: Proceedings of the working conference on reverse engineering (WCRE). IEEE, pp 100–109

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

Roy CK, Cordy JR (2007) A survey on software clone detection research. Tech. Rep. TR 2007-541. Queens University

Roy CK, Zibran MF, Koschke R (2014) The vision of software clone management: past, present, and future (keynote paper). In: 2014 Software evolution week - IEEE conference on software maintenance, reengineering, and reverse engineering, (CSMR-WCRE). IEEE, pp 18–33

Siegmund J, Kästner C, Apel S, Parnin C, Bethmann A, Leich T, Saake G, Brechmann A (2014) Understanding understanding source code with functional magnetic resonance imaging. In: Proceedings of the international conference on software engineering (ICSE). ACM, pp 378–389

Svajlenko J, Roy CK (2014) Evaluating modern clone detection tools. In: 30th IEEE International conference on software maintenance and evolution (ICSME). IEEE, pp 321–330

van Tonder R, Le Goues C (2016) Defending against the attack of the micro-clones. In: 2016 IEEE 24th International conference on program comprehension (ICPC). IEEE, pp 1–4

Trafton JG, Altmann EM, Ratwani RM (2011) A memory for goals model of sequence errors. Cogn Syst Res 12:134–143

Xie S, Khomh F, Zou Y (2013) An empirical study of the fault-proneness of clone mutation and clone migration. In: Proceedings of the 10th working conference on mining software repositories (MSR). IEEE

Zibran MF, Saha RK, Asaduzzaman M, Roy CK (2011) Analyzing and forecasting near-miss clones in evolving software: an empirical study. In: Proceedings of the international conference on engineering of complex computer systems (ICECCS). IEEE, pp 295–304

Moritz Beller is a PhD student at the Delft University of Technology, the Netherlands. His primary research domain is evaluating and improving the feedback developers receive from dynamic and static analysis using empirical methods. Moritz holds an M.Sc. with distinction from the Technical University of Munich, Germany. More on www.inventitech.com.

Andy Zaidman is an associate professor at the Delft University of Technology, The Netherlands. He obtained his MSc (2002) and PhD (2006) in Computer Science from the University of Antwerp, Belgium. His main research interests are software evolution, program comprehension, mining software repositories and software testing. He is an active member of the research community and involved in the organisation of numerous conferences (WCRE'08, WCRE'09, VISSOFT'14 and MSR'18). In 2013 Andy Zaidman was the laureate of a prestigious Vidi career grant from the Dutch science foundation NWO.

Andrey Karpov is technical director of the “Program Verification Systems” company where his task is to develop source code static analyzers. He has worked for several years in the “CFD Software Group” Scientific Center where he has acquired an exceptional experience of resource-intensive software development in the sphere of computational modeling and visualization. It was there that he noticed an insufficient set of tools for detecting defects in 64-bit software handling large memory amounts. It became the starting point in creation of the Viva64 static analyzer and later the PVS-Studio package. Andrey has received six Microsoft MVP awards as a Visual C++ expert and is an Intel Black Belt.

Rolf A. Zwaan is professor of biological and cognitive psychology at Erasmus University Rotterdam. He has published extensively on language, perception, memory, and their interactions.