Version 1.0

How to Achieve Open Source License Compliance

Open Source Software and Reverse Engineering

Karsten Reincke*

March 2, 2015

Sometimes, it is stated that software using LGPL licensed components, may only be distributed by permitting reverse engineering of the software using the LGPL components. By reading the licenses strictly, this article proves in detail that we are not obliged to allow reverse engineering as long as we distribute our open source based work in form of dynamically linkable files. Particularly, the article shows, that we also can compliantly distribute our compiled, but still not linked work without permitting reverse engineering, if it uses LGPL licensed components.

1 Preface

Reverse Engineering is a challenging issue: Sometimes, developers must protect their business relevant work. Thus, they only incorporate open source components published under a permissive or a weak copyleft license: Of course, they want to use the open source software compliantly. But nevertheless, they want be able to distribute their own work as closed software for not exposing their secrets. So, being obliged by the embedded components to permit reverse engineering of the work using the components, would subvert this strategie. Unfortunately, it is often said, that one of the most important weak copyleft licenses, the LGPL, requires to permit reverse engineering.

*) Deutsche Telekom AG, Products & Innovation, T-Online-Allee 1, 64295 Darmstadt

During the last two years, I was repeatedly asked to explain and defend my viewpoint in respect of reverse engineering and open source software: I am profoundly convinced that there is no problem at all as long as we distribute dynamically linkable programs. This position had often astonished my collocutors. Some of them were still looking for an exit of the dilemma, while the others already had given up and could rather believe that there was such a simple solution. So, — at the end of our discussions – they encouraged me to prove my position in detail.

For example, my capable colleague Helene Tamer constantly insisted, that Deutsche Telekom AG could not give up her restrictions to use LGPL libraries until I had offered a reliable proof that the LGPL does not require reverse engineering. Or I could enthousiasticly discuss the permission of reverse engineering with the knowledgable German open source attorney Hendrik Schöttle[1] who boosted me to understand that I had to argue more subtler than I had done in my first answers. And last but not least also Karen Copenhaver[2] considered my arguments and emphatically asked me to share my thoughts with the community.

So, I thankfully[3] can now offer a thoroughly elaborated proof for the assertion that there - in general - is a way to distribute open source software compliantly without permitting reverse engineering, and that this way – in particular – is also usable to compliantly distribute LGPL licensed software without allowing reverse engineering.

Contents

[1] → http://www.osborneclarke.com/lawyers/schottle-hendrik/

[2] → http://www.choate.com/people/karen-copenhaver

[3] Following the spirit of the open source movement, this article is published under the terms of CC-BY-SA 3.0 (→ p. 37). But as a document concerning legal issues, it must be published under a specific proviso: (→ p. 36). And finally, it is first and foremost developed as a chapter of the **O**pen **S**ource **Li**cense **C**ompendium (→ http://www.oslic.org/). But for offering a version which can more simply be distributed, we also produced this extract.

2 Reverse Engineering and Open Source as Challenge

Beyond any doubt, the LGPL mentions "reverse engineering" literally[4] for indicating that reverse engineering in any respect must be allowed to use and distribute LGPL software compliantly:

> *"[. . .] you may [. . .] distribute a work (containing portions of the Library) under terms of your choice, provided that the terms permit [. . .]* reverse engineering *[. . .]"* [5]

There are three strategies for dealing with such provisions: one can try to fully honor its meaning, one can mitigate its meaning, or one can avoid to discuss this requirement altogether:

A first group of well known open source experts take the sentence of the LGPL-v2 as a strict rule which requires that one has to allow reverse engineering of the whole software product if one embeds any LGPL licensed component into that

[4] For the LGPL-v2 cf. *Open Source Initiative*: The GNU Lesser General Public License, version 2.1 (LGPL-2.1); 1999 [n.y. of the html page itself] ⟨URL: http://opensource.org/licenses/LGPL-2.1⟩ – reference download: 2013-03-06, wp., §6; for the LGPL-v3 cf. *Open Source Initiative*: The GNU Lesser General Public License, version 3.0 (LGPL-3.0); 2007 [n.y. of the html page itself] ⟨URL: http://opensource.org/licenses/LGPL-3.0⟩ – reference download: 2013-03-06, wp., §4

[5] cf. *Open Source Initiative*: The LGPL-2.1 License (OSI), 1999, wp, §6. The LGPL-v2 uses the capitalized word "Library" for denoting a library which "[. . .] has been distributed under (the) terms" of the LGPL-v2. (cf. id., l.c., wp, §0). Inside of our LGPL chapter(s) we follow this interpretation.

product[6].

A second group of well known and knowledgeable open source experts signify that the LGPL-v2 indeed literally contains a strict rule, but that this rule actually is not meant as it sounds: For example, two of these experts explain that "these requirements on the licensed combination require that the license chosen not prohibit the customer's modification and reverse engineering for debugging these modifications in the work as a whole". But then they directly add the limitation, that "in practice, enforcement history suggests, it means that the license terms chosen may not prohibit modification and reverse engineering for debugging of modification in the LGPL'd code included in the combination"[7].

---

[6] For example, a very trustworthy German expert states that the LGPL-2.1 generally requires that a distributor of a program which accesses a LGPL-2.1 licensed library, must grant his customer also the right to modify the accessing program and hence also the right to execute reverse engineering. Literally the German text says:

> "Ziffer 6 LGPLv2.1 knüpft die Erlaubnis, das zugreifende Programm unter beliebigen Lizenzbestimmungen verbreiten zu drüfen, an eine Reihe von Verpflichtungen, die in der Praxis oft übersehen werden: Zunächst muss dem Kunden, dem die Software geliefert wird, die Veränderung des zugreifenden Programms gestattet werden und zu diesem Zweck auch ein Reverse Engineering zur Fehlerbehebung. *Dies dürfte alle Formen des Debugging und das Dekompilieren des zugreifenden Programms umfassen.*" (cf. *Jaeger, Till* a. *Axel Metzger*: Open Source Software. Rechtliche Rahmenbedingungen der Freien Software; 3rd edition. München: Verlag C.H. Beck, 2011, pp. 81; emphasis KR).

At first glance, also "copyleft.org" – the "[...] collaborative project to create and disseminate useful information, tutorial material, and new policy ideas regarding all forms of copyleft licensing" (cf. *copyleft.org*: What is copyleft.org; n.l, 2014 ⟨URL: http://copyleft.org/⟩ – reference download: 2014-12-15, wp.) – could be taken as another witness for such an attitude of strict reading: Some of its contributors elucidate in a chapter dealing with "special topics in compliance" that "the license of the whole work must [sic!] permit 'reverse engineering for debugging such modifications' to the library" and that one therefore " should take care that the EULA used for the Application does not contradict this permission"(cf. *Kuhn, Bradley M.* et al.: Copyleft and the GNU General Public License: A Comprehensive Tutorial and Guide; n.l, 2014 ⟨URL: http://copyleft.org/guide/comprehensive-gpl-guide.pdf⟩ – reference download: 2014-12-15, p. 86

[7] cf. *Moglen, Even* a. *Mishi Choudhary*: Software Freedom Law Center Guide to GPL Compliance, 2nd Edition; 2014 ⟨URL: https://www.softwarefreedom.org/resources/2014/SFLC-Guide_to_GPL_Compliance_2d_ed.html⟩ – reference download: 2014-12-15, wp., chapter LGPLv2.1, section 6. Such a mitigation can also be found in the tutorial of copyleft.org: After they have summarized the LGPL-v2 sentence as a strict rule, they directly continue, that one "[. . .] must refrain from license terms on works based on the licensed work that prohibit replacement of the licensed components of the larger non-LGPL'd work, or prohibit decompilation or reverse engineering in order to enhance or fix bugs in the LGPL'd components" (cf. *Kuhn* et al.: Copyleft and the GNU General Public License, 2014, p. 86). This added specification indicates, that one only has to facilitate the modification of the library and that reverse engineering can be ignored as long as there are other ways to improve the embedded library.

Finally, a third group of experts prefers not to discuss the problem of reverse engineering, although this technique is literally mentioned in the license and although they want explain how to use GPL/LGPL licensed software compliantly[8].

This situation must bother companies and people who want to use open source software compliantly and who therefore are looking for guidance. Particularly it disturbs those who want to protect their business relevant software. At the end, they might consider that this sentence is not consistently understood by the open source community itself. And – as far as we know – at least some of these companies preventively prohibit their developers to embed LGPL licensed components into programs which contain business relevant techniques. Unfortunately, this consequence does not only obstruct the access to a large set of well written free software, but it is scarcely possible to obey such an interdiction consequently: The glibc, which enables the programs to talk with the kernel of the GNU/Linux system[9], is licensed under the LGPL[10]. And hence, this library is indirectly linked to or combined with any program running on the GNU/Linux system. So, if the LGPL-v2 indeed required, that reverse engineering of every program must be allowed, which contains portions of any LGPL Library, then every GNU/Linux user would be allowed to examine every program running on GNU/Linux by *reverse engineering*, simply, because finally every 'GNU/Linux program' is linked to or combined with the glibc[11]. In other words: if the LGPL indeed required the permission of reverse engineering, then every program executed on GNU/Linux may be reverse engineered.

But an exhaustive reading of the LGPL-v2 strongly indicates, that there must be another valid, more 'liquid' understanding of the LGPL: The preamble explains

[8] An article of Terry J. Ilardi might be taken as a first witness of this third strategy: he profoundly explains the essence of the LGPL, he especially discusses §6, and he delivers applicable rules like "DO NOT statically link to LGPL [...] code if you wish to keep your program proprietary". But he does not discuss *reverse engineering* (cf. *Ilardi, Terry J.*: Common OSS License Problems; n.l, 2010 ⟨URL: http://www2.aipla.org/html/spring/2010/papers/Ilardi_Paper.pdf⟩ – reference download: 2014-12-16, pp. 5f). Similarily argues Rosen (cf. *Rosen, Lawrence*: Open Source Licensing. Software Freedom and Intellectual Property Law; Upper Saddle River, New Jersey: Prentice Hall PTr, 2005, ISBN 0–13–148787–6, pp. 121ff). And – despite their comments on reverse engineering in the specific chapter *special topics in compliance* – the copyleft.org document can also be taken as an instance of this attitude: Although its' authors recommend to "study chapter 10 carefully" for establishing an adequate "compliance with LGPLv2.1" (cf. *Kuhn* et al.: Copyleft and the GNU General Public License, 2014, p. 86), this chapter 10 – dedicated to the meaning of the "Lesser GPL" – does not deal with reverse engineering, although it discusses the §6 of the LGPLv2.1 in depth (cf. id., l.c., pp. 56ff, esp. 60f).

[9] cf. http://www.gnu.org/software/libc/

[10] cf. http://en.wikipedia.org/wiki/GNU_C_Library

[11] This conclusion might surprise. But it is inferred with exactly the same arguments as the conclusion, that without a licence offering a weaker copyleft every program would have been licensed under the GPL. The copyleft.org document explains this argumentation in great detail (cf. id., l.c., pp. 56f).

the reason for offering another weaker license beside the GPL. It says that "[...] on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard" and that therefore it could be strategicly necessary to "allow [...] non-free programs [...] to use the library" without enforcing that these programs become free software too[12].

So, if the LGPL had indeed determined that every program linked to or combined with any LGPL library may be reverse engineered, then the LGPL would have an effect contrary to its own intention. It would have introduced something like *'security by obscurity'*: First, the LGPL would allow to protect the internals of your own work against investigation by allowing to keep the code of the non-free program using the library a scecret[13]. But then – in the end – the LGPL would also allow the user to reverse engineer the received binary and hence would enable him to nevertheless discover all internals[14]. Hence, finally the LGPL-v2 would undermine its' own raison d'ètre introduced by its' inventors: under such circumstances there probably would have been less hope that any LGPL library could have become a defacto standard.

We know that the inventors of the GNU licenses and GNU software are very sophisticated experts. They never would have published such an inconsistent document. So, this dissent read in(to) the document is a strong indicator for the fact, that there must be a better way to understand the license. And thus, it is up to us, the followers, to explicate a more adequate interpretation. Of course, such an interpretation must be grounded on the written text. No doubt: we, the scholars, are not allowed to add our own wishes. We must read the license very strictly. We have to deduce 'understandings' only by matching the interpretations explicitly and reasonably back to the license text itself.

Encouraged by the indication that a better understanding of the LGPL may exist and contrary to the other strategies, we are going to prove that there is a valid way to compliantly distribute any open source based software without permitting reverse engineering: We want to show that none of the main open source licenses[15] requires reverse engineering if the work using the open source

[12] cf. *Open Source Initiative*: The LGPL-2.1 License (OSI), 1999, wp, §preamble.

[13] The weak copyleft has been introduced for encouring the widest possible use of the library.

[14] It would only cost a little more effort - as security by obscurity indicates.

[15] Just as the OSLiC, also this part focuses only on the most important open source licenses (cf. https://www.blackducksoftware.com/resources/data/top-20-open-source-licenses wp.): the Apache license (cf. *Open Source Initiative*: Apache License, Version 2.0; 2004 [n.y. of the page itself] ⟨URL: `http://opensource.org/licenses/Apache-2.0`⟩ – reference download: 2013-02-07, wp.), the BSD licenses (cf. *Open Source Initiative*: The BSD 3-Clause License; 2012 [n.y.] ⟨URL: `http://www.opensource.org/licenses/BSD-3-Clause`⟩ – reference download: 2012-07-04, wp. and cf. *Open Source Initiative*: The BSD 2-Clause License; 2012 [n.y.] ⟨URL: `http://www.opensource.org/licenses/BSD-2-Clause`⟩ – reference download: 2012-07-03, wp.)), the MIT license (cf. *Open Source Initiative*: The MIT License; 2012 [n.y.] ⟨URL: `http://opensource.org/licenses/mit-license.php`⟩ – reference download:

Library is distributed in form of dynamically linkable files. In particular, we are going to prove that one even has not to allow reverse engineering of the work using an LGPL Library, if one distributes that work as dynamically linkable files. And we want to show that in all other cases one has at least to fear that one has implicitly allowed the reverse engineering of the work using the open source Library if one distribute that work. In particular, we want to prove that one has to fear this implicitly given permission even if one distributes a work using a library licensed under any permissive license[16].

In general, we hope that our analysis, grounded on the license text itself, will support companies and people to compliantly use open source software more often and with less scruples.

Hence, let us prove our position 'bottom up'. Let us firstly show that it is true for the LGPL-v2 – by explicating the license text lingually, then logically, and finally empirically, before we infer the correct understanding. Then let us show that it is also true for the LGPL-v3. And in the end let us show that it is true for all other licenses[17].

2012-08-24, wp.), the MS-PL (cf. *Open Source Initiative*: Microsoft Public License (MS-PL); 2013 [n.y.] ⟨URL: http://opensource.org/licenses/MS-PL⟩ – reference download: 2013-02-26, wp.), the PostgreSXQL (cf. *Open Source Initiative*: The PostgreSQL Licence (PostgreSQL); 2013 [n.y.] ⟨URL: http://opensource.org/licenses/PostgreSQL⟩ – reference download: 2013-02-27, wp.), the PHP license (cf. *Open Source Initiative*: The PHP License 3.0 (PHP-3.0); 2013 [n.y.] ⟨URL: http://opensource.org/licenses/PHP-3.0⟩ – reference download: 2013-02-27, wp.), the EPL (cf. *Open Source Initiative*: Eclipse Public License, Version 1.0; 2005 [n.y. of the page itself] ⟨URL: http://opensource.org/licenses/EPL-1.0⟩ – reference download: 2013-02-20, wp.), the EUPL (cf. *Open Source Initiative*: European Union Public License, version 1.1 (EUPL-1.1; 2007 [n.y. of the html page itself] ⟨URL: http://opensource.org/licenses/EUPL-1.1⟩ – reference download: 2013-03-04, wp.), the MPL (cf. *Open Source Initiative*: Mozilla Public License 2.0 (MPL-2.0); 2013 [n.y.] ⟨URL: http://opensource.org/licenses/MPL-2.0⟩ – reference download: 2013-02-07, wp.), the LGPLs (cf. *Open Source Initiative*: The LGPL-2.1 License (OSI), 1999, wp. and cf. *Open Source Initiative*: The LGPL-3.0 License (OSI), 2007, wp.), the GPLs (cf. *Open Source Initiative*: GNU General Public License, version 2 (GPL-2.0). Version 2, June 1991; 1991 [n.y. of the html page itself] ⟨URL: http://opensource.org/licenses/GPL-2.0⟩ – reference download: 2013-02-05, wp. and cf. *Open Source Initiative*: GNU General Public License, version 3 (GPL-3.0); 2007 [n.y. of the html page itself] ⟨URL: http://opensource.org/licenses/GPL-3.0⟩ – reference download: 2013-03-05, wp.) and the AGPL (cf. *Open Source Initiative*: GNU Affero General Public License, Version 3 (AGPL-3.0); 2007 [n.y. of the html page itself] ⟨URL: http://opensource.org/licenses/AGPL-3.0⟩ – reference download: 2013-04-05, wp.)

16) By the way, our analysis should also provide proof that the LGPL is not something like a 'poisoned' license containing "an imprenetrable maze of technology babble" which "[...] should not be in a general-purpose software license" (cf. *Rosen*: Open Source Licensing, 2005, p. 124). The challenge of the today's descendants is to understand the former inventors of the GNU licenses and their way to think about computing - including all the hassle the computing language C might provoke.

17) analysed by the OSLiC: → p. 6.

3 Reverse Engineering in the LGPL-v2

The LGPL-v2.1 contains one sentence which literally refers to the issues of *reverse engineering*:

> *"[...] you may [...] combine or link a 'work that uses the Library' with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and* reverse engineering *for debugging such modifications."*[18]

Hereinafter, we will sometimes denote these lines by the term *LGPL2-RefEng-Sentence*.

3.1 Linguistical Clarification

For fulfilling our rule, to read the text strictly and deduce our interpretations reasonably, let us firstly only highlight the syntactical conjunctions for simplifying the understanding:

> *"[...] you may [...] combine* ***or*** *link a 'work that uses the Library' with the Library to produce a work containing portions of the Library* ***and*** *distribute that work under terms of your choice,* ***provided that*** *the terms permit modification of the work for the customer's own use* ***and*** reverse engineering *for debugging such modifications."*[19]

It is evident that the conjunction *'provided that'* is splitting the sentence into two parts: you are allowed to do something *provided that* a condition is fulfilled. Additionally, both parts of the sentence – the one before the conjunction *'provided that'* and the part after it – are syntactically condensed embedded phrases which also contain subordinated conjunctions and elliptical constructions[20]. These syntactical interconnections must be disbanded:

Let us firstly **dissolve the syntactical compression before the conjunction *'provided that'***: It is established by using the two other conjunctions *and* and *or* and introduced by the subordinating phrase *you may [...]*. Unfortunately, from a formal point of view, one can read the phrase *you may (X or Y and Z)* as

[18] cf. *Open Source Initiative*: The LGPL-2.1 License (OSI), 1999, wp., §6.. The first ellipse in this citation – notated by the string '[...]' – refers to the phrase "As an exception to the Sections above,", the second to the phrase "also". These words together want to indicate, that the LGPL offers its §6-way-of-distribution as an exception to the intended default way of distributing such a Library. So, the nature of the extraordinary way itself is not affected by this hint. Thus, we feel free to erase this contextual link.

[19] cf. id., l.c., wp., §6, emphasis KR..

[20] cf. http://en.wikipedia.org/wiki/Ellipsis_%28linguistics%29, wp.

two different groupings: either as *you may ((X or Y) and Z)* or as *you may (X or (Y and Z))*.

But, fortunately, we know from the semantic point of view that speaking about *"[...] combining **or** linking [...something] to produce a work containing portions of the Library"* denotes two different methods which both can *join* the components *"[...] to produce a work containing portions of the Library"*. So, let us – only for a moment[21] – simply replace the string *"combine or link"* by the string *"*join"*[22]. This reduces the syntactical structure of the sentence back to the simple phrase *you may (W and Z)* in which *W* stands for *(X or Y)*.

Now, we can directly state that the phrase *you may (W and Z)* itself is a condensed version of the explicit phrase *(you may W) and (you may Z)*.

Finally we have to note, that the phrase before the conjunction *'provided that'* contains also a linguistic ellipsis[23]: It says that you may *join the components "to produce **a work containing portions of the Library and** distribute **that work** under terms of your choice". With respect to the English grammar, we may conclude that the second term *that work* refers back to the previously introduced specification of *a work containing portions of the Library*: if a complete phrase has just been introduced explicitly, then the English language allows to reduce its' next occurence syntactically while its' complete meaning is retained. Hence, conversely, we are allowed to unfold the reduced form to restore the complete phrase.

So – overall – we may understand the phrase before the conjunction *'provided that'* as a phrase with the structure *(you may W) and (you may Z')*:

> ***((**you may [...]* *join *a "work that uses the Library" with the Library to produce a work containing portions of the Library**) and (**you may [...] distribute that work containing portions of the Library under terms of your choice**))*** **provided that** [...]

Theoretically, a reader could reject our first dissolution of the LGPL2-RefEng-Sentence. But for reasonably denying our interpretation he has to deliver other resolutions of the lingustic elliptical subphrases or other dissolvations of the conjunctions. Fortunately, it seems to be evident that such attempts must violate the English grammar.

Let us secondly **dissolve the part after the conjunction *'provided that'***: With respect to the subordinated conjunction *'and'*, the subphrase *the terms permit*

[21] Later on we will re-insert th original phrase!

[22] When the LGPL and the GPL were initially defined, the C programming language was the predominant model of software development. Knowing this method eases the understanding of these licenses. Thus, it is not totally wrong to take this token *join also as a curtsey to the C programming language.

[23] cf. http://en.wikipedia.org/wiki/Ellipsis_%28linguistics%29, wp.

syntactically refers to both, the *modification* and the *reverse engineering*: An embedded conjunction *'and'* allows to use a more stylish grammatical compaction. So, it should be clear, that saying

> **provided that** *the terms permit modification of the work for the customer's own use* **and** *reverse engineering for debugging such modifications*

means

> **provided that** *the terms permit (modification of the work for the customer's own use* **and** *reverse engineering for debugging such modifications)*

and is totally equivalent to the sentence

> [...] **provided that** *((the terms permit modification of the work for the customer's own use)* **and** *(the terms permit reverse engineering for debugging such modifications)).*

We believe that there is no other possibility to understand this part of the LGPL2-RefEng-Sentence with respect to the rules of the English language. Nevertheless, this is a next point where our reader may formally disagree with us. If he really wants to object our dissolution, he must deliver another valid interpreation of the scope of the conjunction *and* or he must deliver another resolutions of the linguistic ellipsis. But we reckon, that one can not reasonably argue for such alternatives.

Finally, there are other deeply embedded ellipses, which need to be resolved as well:

1. In the part before the splitting conjunction *'provided that'* we already had to expand the abridging *'that work'* by its intended explicated version *'that work containing portions of the Library'*. In the part after the splitting conjunction the first subphrase also contains the term *'the work'*. Formally, this term can either refer to *'the work that uses the library'* as one of the components which are joined, or it can refer to *'the work containing portions of the Library'* as the result of joining the components. We decide to constantly dissolve the elliptic abridgement by the phrase *'the work containing portions of the Library'*.
2. The first clause of the part after the splitting conjunction *'provided that'* talks about the purpose of "permitting modification of the work" which we just had to unfold to the phrase *'permitting modification of the work containing portions of the Library'*. The second clause talks about the purpose of "permitting reverse engineering": it shall support the "debugging [of] such modifications". The pronoun *'such'* indicates that the word *'modifications'*

refers back to the just unfolded phrase *modification of the work containing portions of the Library*. So, even the second sentence has to be expanded to that explicit phrase.

3. Finally and only for being complete, we also have to unfold the clause "the terms" to the form which is predetermined by the first referred instance "the terms of your choice"

So – overall – we are allowed to rewrite the LGPL2-RevEng-Sentence in the following form, namely without having changed its meaning[24]:

```
( ( you may
       *join a work that uses the Library with the Library
        to produce a work containing portions of the Library )
  AND
  ( you may
        distribute that work containing portions of the Library
        under terms of your choice
) )
PROVIDED THAT
( ( the terms of your choice permit
        modification of the work containing portions of
        the Library for the customer's own use )
  AND
  ( the terms of your choice permit
        reverse engineering for debugging modifications
        of the work containing portions of the Library
) )
```

At this point we must recommend all our readers to verify that this 'structurally explicated presentation' does exactly mean the same as the intially quoted LGPL2-RefEng-Sentence. We are now going to discuss some of its' logical aspects by some formal transformations. For accepting these operations and linking the results back to the original LGPL2-RefEng-Sentence, it is very helpful to know that one already has accepted the equivalence of this explicated form and the more condensed original version. For reviewing the equivalence the reader could – for example – ask himself which of our rewritings are wrong, why they are wrong and which alternatives can reasonably be offered for solving the syntactical issues which disposed us to chose our solutions. Again, we ourselves – of course – are profoundly convinced that both versions are completely equivalent.

[24] Recollect that '*join' still stands for 'combine or link'.

3.2 Logical Clarification

For simplifying our discussion let us now replace the meaningful terminal phrases of our form by some logical variables:

Γ :- (you may *join a work that uses the Library with the Library to produce a work containing portions of the Library)

Δ :- (you may distribute that work containing portions of the Library under terms of your choice)

Φ :- (the terms of your choice permit modification of the work containing portions of the Library for the customer's own use)

Σ :- (the terms of your choice permit reverse engineering for debugging modifications of the work containing portions of the Library)

Θ :- Γ *and* Δ

Ω :- Φ *and* Σ

Based on these definitions, we can syntactically reduce the LGPL2-RefEng-Sentence to the formula (Γ *and* Δ) *provided that* (Φ *and* Σ) or – even shorter – to (Θ *provided that* Ω).

Now, we have to clarify the meaning of the conjunction *'provided that'*:

Obviously, *provided that* means something like *under the condition that*. So, one might try to take this conjunction as another more stylish version of the common *if(...)then(...)*-formula, sometimes also identified as a (logical) implication[25]. Thus, we have to consider the process of sequencing the linguistic form into a logical formula: if we indeed take the conjunction *provided that* as another form of the logical implication, it is not evident, which part of the linguistic sentence must become the premise, and which the conclusion: Does *(Θ provided that Ω)* mean *(if Θ then Ω)* or *(if Ω then Θ)*?

Apparently, *provided that* wants to establish something like a precondition. So, one might conclude that (Θ *provided that* Ω) means *(if Ω then Θ)* or – more logically notated – $((\Phi \wedge \Sigma) \rightarrow (\Gamma \wedge \Delta))$. If this interpretation is adequate, it must of course fulfill the intended purpose of the corresponding LGPL-v2-section, which wants to regulate the distribution of works containing portions of LGPL libraries.

For facilitating the understanding of our argumentation, let us first check whether this logical interpretation of the linguistic conjunction fits the purpose of the LGPL

[25] Actually the logical implication and the computational if-then-construct are not equivalent. Fortunately, we later on can show, that in the context of this discussion the difference can be ignored.

– by unfolding the slightly reduced version ($\Sigma \rightarrow \Delta$) back to the corresponding verbal form:

> ***if (*** *[...] the terms permit reverse engineering for debugging modifications of the work containing portions of the Library,* ***) then (*** *[...] you may distribute that work containing portions of the Library under the terms of your choice.)*

Now we can better see the problem: An implication as a whole is false only if the premise is true and the conclusion is false. In all other cases it is true. Especially, it is true, if the premise is false: If the premise is false, then the truth value of the conclusion does not matter in any sense. Thus, if we take this implication as a rule, which shall determine our behaviour, then this implication only supports us, if we already have decided to permit reverse engineering. In this case the rule successfully tells us that we are allowed to distribute the work containing portions of the Library. But from the converse decision that we will not permit reverse engineering, follows nothing - because a false premise does not influence the truth value of the conclusion. Especially, the rule does not tell us that we may not distribute the work containing portions of the Library. So – from the viewpoint of the formal logic – this translation of the original conjunction *'provided that'* says, that if the terms of your own license do not permit reverse engineering for debugging modifications of the work containing portions of the Library[26], then **you may or may not** distribute that work containing portions of the Library under the terms of your choice[27]. Hence, we must state that this interpretation does not fulfill the purpose of the LGPL-V2: if reverse engineering is not allowed, the distribution of the work containing portions of the Library is not regulated. We have to conclude, that this sequencing the LGPL2-RefEng-Sentence as a logical implication is wrong.

But we deduced this consequence from a slighty reduced form of the LGPL2-RefEng-Sentence. Thus, we still have to ask, whether we have to derive this conclusion also on the base of the completely unfolded formula ($(\Phi \wedge \Sigma) \rightarrow (\Gamma \wedge \Delta)$)? The answer is yes: the premise ($(\Phi \wedge \Sigma)$ contains a logical conjunction. So the truth value of the whole premise depends on the truth value of each of its terminal statements, particularly on that of the statement Σ: If we decide not to permit reverse engineering, then the premise as whole is false, regardless we forbid or allow modifications. Consequently, the premise does not influence the truth value of the conclusion. So, there is no way, to conclude that we have to allow or that we do not have to allow reverse engineering. Hence we can transfer our result, deduced for the slightly reduced formula to the unfolded complete formula: assuming that (Θ *provided that* Ω) means *(if* Ω *then* Θ) is wrong.

---

[26] The premise is false.

[27] The truth value of the conlusion is undetermined by the rule.

So, let us test the other combination. Let us ask, whether *(*Θ *provided that* Ω*)* means *(if* Θ *then* Ω*)* or – more logically notated – $((\Gamma \wedge \Delta) \rightarrow (\Phi \wedge \Sigma))$. If we again for a moment focus on the reduced version $(\Delta \rightarrow \Sigma)$ and dissolve our replacements, then we get back the rule:

> ***if*** *(* *[...] you may distribute that work containing portions of the Library under the terms of your choice,* *)* ***then*** *(* *[...] the terms permit reverse engineering for debugging modifications of the work containing portions of the Library.* *)*

Now we can see, that this version perfectly regulates the distribution of works containing portions of LGPL libraries: If we are allowed to do so or – in other words: if we are compliantly distributing works containing portions of LGPL libraries[28], then we have to permit reverse engineering[29]. This follows from applying *Modus Ponens* to the implication[30]. And if we do not permit reverse engineering[31], then we are not allowed to distribute works containing portions of LGPL libraries[32]. This follows from applying *Modus Tollens* to the implication[33]

But – again – we have to consider that we have deduced this consequence from a slighty reduced version of our LGPL2-RefEng-Sentence. Thus, we still have to show that our result also holds for the completely unfolded formula $((\Gamma \wedge \Delta) \rightarrow (\Phi \wedge \Sigma))$: If we want to distribute works containing portions of the Library which have been produced by joining the Library and the work using the Library[34], then our terms must permit the modification *and* reverse engineering of the distributed product[35]. And if we do not allow its modification *or* reverse engineering[36], then we do not compliantly distribute works containing portions of the Library which have been produced by joining the Library and the work using the Library[37] Thus, we may generally state, that the logical explication $((\Gamma \wedge \Delta) \rightarrow (\Phi \wedge \Sigma))$ perfectly regulates the distribution of works containing portions of LGPL libraries.

Based on this clarification, we can reasonably replace the more stylish conjunction *'provided that'* by its more known equivalent *'implication'*[38], which we indicate by

---

[28] The premise is true.

[29] The conclusion must be true, too!

[30] A true premise evokes a true conclusion based on the given truth of the implication / rule itself.

[31] The conclusion is false.

[32] The premise must be false, too!

[33] A false conclusion evokes a false premise based on the given truth of the implication / rule itself.

[34] Premise is true.

[35] Conclusion must become true by Modus Ponens.

[36] Conclusion is false.

[37] Premise must become false by Modus Tollens.

[38] Here we can also see, that the difference between the if-then-command as part of a procedural computer language and the logical implication does not influence our results: In the context

the commonly used character for a logical implication, the sign '$\rightarrow$':

Θ provided that Ω

$\equiv \Theta \rightarrow \Omega$

$\equiv (\Phi \wedge \Sigma) \rightarrow (\Gamma \wedge \Delta)$

$\equiv$ (([Φ] you may
*join a work that uses the Library with the Library
to produce a work containing portions of the Library)
$\wedge$
([Σ] you may
distribute that work containing portions of the
Library under terms of your choice
))
$\rightarrow$
(([Γ] the terms of your choice permit
modification of the work containing portions of
the Library for the customer's own use)
$\wedge$
([Δ] the terms of your choice permit
reverse engineering for debugging modifications
of the work containing portions of the Library
))

3.3 Empirical Clarification

We can now simplify this formula once more by considering some empirical facts and explicating some underlying understandings:

The first sentence Φ explains that the *work that uses the Library* and the used *Library* itself together are joined and thereby transformed into a *work containing portions of the Library*. So, formally, one might ask, whether this newly generated *work containing portions of the Library* also still *uses the Library*?

Unfortunately, it is empirically possible, that such a process for combining the two components could (a) copy all original portions of the library into a something like a 'dead end section' of the program where they are never excuted, and could (b) replace all original portions of the library by functionally equivalent portions

of a procedural if-then-command the truth of the premise triggers the execution of the conclusion. In our discussion, this aspect is totally covered by the Modus Ponens derivation of the logical interpretation. And the Modus Tollens derivation of the logical interpretation on the other side does not play any role in a procedural if-then-command. So, it was the right decision to understand the LGPL2-RefEng-Sentence logically and not as procedual command.

of any other library. Thus, the resulting *work containing portions of the Library* would indeed still contain portions of the Library, although it would not use it any longer. And because of this possibility, we are not allowed to say, that every work containing portions of a library also uses the library[39].

But, fortunately, the normal computational process of *combining and linking a work that uses the Library with the Library to produce a work containing portions of the Library* inherently preserves the utilization of the joined library: It is the general purpose of a software library to offer functions and/or data (structures) for really being used by applications. And vice versa, software developers refer to a specific library because they prefer its service: They use readily prepared libraries (or classes or anything else) because they want to simplify their own work while they conserve the quality level of their work. Thus, they chose a library based on the assertion, that the standard compiling and linking process guarantees, that indeed the chosen library is used (and not secretly substituted by a mysterious 'equivalent'). With respect to this praxis of programming we are allowed to say that a *work containing portions of the Library* which has been **built by the normal development processes** of combining, compiling, and linking source and object files, indeed also uses the intended library.

Now, we are able to consider an empirical correlation between the first sentence Φ and the second sentence Σ:

It seems to be evident, that we must already have done Φ, in other words: that we must already have **joined – respectively: combined or linked – a work that uses the Library with the Library to produce a work containing portions of the Library*, if we are going to compliantly *distribute that work containing portions of the Library under terms of your choice*. Or briefly spoken: It seems to be conclusive that Σ ***empirically*** *implies* Φ[40].

But is this conclusion correct? Let us check this statement by assuming the opposite: If the contrary was true, there had to exist a *work containing portions of the Library* which had been gained without having linked or combined the work and the Library in any sense. But from the inference above we already know that *works containing portions of the Library*, which have been produced by the standard computational processes of *combining and linking a work that uses the Library with the Library*, indeed also *use the Library*. Thus, it would be self-contradictory to talk about a *work containing portions of the Library*, which was produced by the standard combining and linking processes, and similarily to state, that exactly this work is not combined with the library in any sense. And from a proof by contradiction we may infer the truth of the logical opposite:

So, with respect to the meaning of *being standardly combined or linked with*, we

[39] ... even if we think that this is a really silly way to organize the joining process!

[40] but not vice versa.

may now say, that

- it is necessarily true that a computional work, which is standardly produced on the base of *a work that uses the Library* and *the Library* and which therefore literally contains more or less *portions of a library*, indeed uses the *the Library* and *is* therefore *combined with the library*.
- Σ[41] empirically implies Φ[42] (in the standardized world of software development), because Φ must ever have been executed when Σ is going to be realized.

Thus, we can now reduce the LGPL2-RefEng-Sentence to its' real core, the LGPL2-RefEng-Rule:

```
(   [Σ] you may
        distribute (a) work containing portions of the
        Library[43] under terms of your choice )
→
( ( [Γ] the terms of your choice permit
        modification of the work containing portions of
        the Library for the customer's own use )
  ∧
  ( [Δ] the terms of your choice permit
        reverse engineering for debugging modifications
        of the work containing portions of the Library
) )
```

This is indeed the essence of the LGPL2-RefEng-Sentence. It logically explains us that we have to *allow reverse engineering* and modification of a *work containing portions of the Library* if we distribute it (Modus Ponens) and that we are *not allowed to distribute a work containing portions of the Library*, if we do *not allow* its modification or *reverse engineering* (Modus Tollens).

Thus, for applying this rule correctly, we now only must know whether a work indeed contains portions of the Library or not.

3.4 Final Conclusion

Unfortunately, there are more than one software developing scenarios, which must be considered for answering this question in detail. We see three general types of

[41] distributing *a work that uses the Library and contains portions of a library*

[42] A work that uses the Library has been *joined with the Library to produce a work containing portions of the Library

[43] which previously has been prepared for being distributed by standardly combining and linking the work that uses the Library with the Library in a way that this prepared work indeed also uses the Library

developing computer software:

1. You can produce software by using script languages. Source files which contain script language commands are distributed and executed by an interpreter without priorly being transformed into another 'more' machine specific language.
2. You can develop software by using languages which are designed for being compiled into a machine independent bytecode. Later on, this independent bytecode is executed by a machine specific virtual machine.
3. You can write traditional software files. Sometimes, these files are remastered by a preprocessor before the real process starts. The traditional sources themselves or the output files of the preprocessor are then compiled and linked as machine specific binary file(s).

You may take 'php' is an example for the first environment, 'Java' an example for the second, and 'C/C++' an example for the third.

Fortunately, the nature of these environments simplifies the answer to the question under which conditions the work using the Library contains portions of the Library:

3.4.0.1 Distributing works with manually copied portions of the Library evokes the copyleft effect: Manually copying code from the sources of the Library into the overarching work that uses the Library, is not the standard way of combining both components, neither in the world of script programming, nor in the world of bytecode programming, nor in the world of programming machine specific code:

Normally, the work which uses the Library is joined to the intended Library by an include statement, an input statement, an import statement, a package statement, or anything else. These *join-statements are inserted into the code of the work. They denote the file(s) which deliver(s) the used functions, methods, classes, or data. It is an integrated feature of the normal development tools that inserting such *join-statements does not directly augment the work using the library by some code of the Library: The development processes are designed to offer an automatic augmention as part of the standard compilation which is started after the actual development loop has been terminated.

Nevertheless, developers can circumvent these standard methods for using a Library. Technically, they can directly copy code from the Library into their own work. Consequently, these manually copied extensions of the code will be compiled and/or executed together with the 'own' code of the work. Thus, it is clear, that in this case the work that uses the Library, already contains portions of the Library, paticularly before the normal *join-processes of the environment are executed.

Hence, if you are going to distribute works that contain literal copies of the Library source code, then you have to allow reverse engineering, even if they have already been compiled (but still not linked) on the base of such augmented files[44].

But, if we manually copy code from the Library to our work using the Library, we also have to consider, that the LGPL-v2 directly regulates this kind of using the Library: It says, that "you may modify your copy or copies of the Library or any portion of it [...] provided that you also [...] cause the whole of the work to be licensed [...] under the terms of this License"[45]. Thus, there are strong arguments for the proposition, that the LGPL causes the copyleft effect in case of literally copying code from the Library into the work using the Library: The code of the work using the library has to be made accessible, as well.

So, overall, we might say, that 'manually' copying code from an LGPL-v2 Library

[44] This directly follows from the LGPL2-RefEng-Rule by Modus Ponens. But nevertheless, one might reply here, that even the result of manually copying code from the Library to the work using the Library is covered by the limits of tolerance, introduced by the LGPL-v2-§5. Formally, this argument seems to be appropriate. And indeed, also we have to consider these limits of tolerance later on. But in the context of copying code from the Library into the work manually, a closer look reduces its impact. You have to discriminate three cases:

1. Developers can **manually copy / transfer some or at most all elements of the Library header files into the code of the work** which the preporcessor itself would copy / transfer into that code automatically. But developers will not do that. Some simple include commands would cause the same effect. And Developers want to save recourses, especially their own working time. So, why should they manually do, what they can delegate to the standard development process. Thus, it is reasonable to assume, that developers, who nevertheless copy portions from Library into their work, do not want to repeat the service of the preprocessor manually, but to transfer more than only these elements. Hence, it is reasonable to assume, that their work is covered by the LGPL2-RefEng-Rule.

2. Developers can **manually copy / transfer more than only the elements of the Library header files from the Library sources into the code of the work using the Library and they can nevertheless let the work being linked to the Library**. But again developers will not do that, because – again – some simple include and linking commands would cause the same effect. So it is reasonable to start from the premise, that copying developers in fact do more than this. Thus, it is reasonable to assume, that their work is covered by the LGPL2-RefEng-Rule.

3. Developers can **manually copy / transfer more than only the elements of the Library header files from the Library sources into the code of the work using the Library without linking it to the Library**. This is a reasonable step of work, because it spares the developers to link their work to the Library. But – by definition – such an augmented work contains more elements of the Library than LGPL-v2-§5 tolerates. Thus it is – again – reasonable to assume, that such a work is covered by the LGPL2-RefEng-Rule.

Hence – overall and from a practical point of view – we can indeed say, that manually copying code from the Library into the work using the Library requires to allow reverse engineering.

[45] cf. *Open Source Initiative*: The LGPL-2.1 License (OSI), 1999, wp., §2, escpcially §2c.

into a work using that Library as a bypass of the standard software combining processes and distributing the result indeed requires to additionally permit its reverse engineering – even if this permission is probably not very important for the recipient, because he probably must have a direct access to the code.

3.4.0.2 Distributing scripts does not need reverse engineering: Computer programs written in a script language are distributed as they have been developed. They are not transformed into another kind of code[46]. The interpreter takes the script file as it is and directly executes it. Thus, there is no special technique of reverse engineering for understanding these kind of software: you can directly read it if you know the script language.

So, again, we might conclude, that a script using a script Library perhaps requires to permit its reverse engineering – but probably this permission is not very important for the recipient, because he can directly read the code.

3.4.0.3 Distributing statically combined bytecode requires the permission of reverse engineering: In Java – the prototype for languages which are compiled to machine independent portable bytecode – each class is compiled as a seperate class file. These class files have to be stored somewhere in the classpath. A side from that, classes can also be collected and distributed in form of packages which then can be used like 'traditional' Libraries. These packages must also be stored somewhere in the classpath. A single class is made known to the work, that want to use it, by an import statement which contains the class name; a whole Java library is made usable by integrating a package statement into the code.

The code which follows such import- or package statements, can then use the definitions offered by the classes. It denotes the elements of the classes by the (qualified) names of its public or protected member variables or methods. Thus, – from a strict viewpoint – the code of such a Java work using a Library indeed contains portions of that library, even if these portions are only identifying names or data structures containing identifying names. The Java compilation process which generates the bytecode, preserves these denoting names. It does not replace the referring names by the referred code of the methods and so on. Only just at the end, when the java virtual machine itself tries to execute the work using the Library, it collects all necessary commands of all 'joined' classes.

So, one might tend to argue that answering the question, whether a distributed java bytecode already contains portions of the used Library, depends on the

[46] Java script is often offered as compressed code. Roughly spoken, this means that at first all white space signs have been replaced by blanks and then all rows of blanks have been reduced to at most one single blank. So, even then, the code itself is directly readable and comprehensible – even if only for very sophisticated experts.

interpretation, whether a denoting identifier of a Library indeed is a portion of the Library. We will discuss this case together with the corresponding C/C++-Case.

But there is another Java specific aspect, which has to be considered as well. As already mentioned, in Java you can also join your work containing the denoting indentifiers and the denoted Library by building a new package, which then contains both, the work using the Library and the used Library. Hence, one can say, that this package is quasi statically linked: if you distribute such an integrated package, then you are distributing both components together. Thus, if you distribute a complete package, in other words: a quasi statically linked work containing the work using the Library and (all portions of) the Library, then you have to permit reverse engineering[47].

So, preliminarily we conclude, that with respect to Java programming you (a) have to permit reverse engineering, if you distribute your work using the Library and the Library itself as a (statically linked) integrated package[48] and that (b) in all other cases your obligation to permit reverse engineering depends on the interpretation whether the identifiers declared by a Library are indeed portions of the Library.

Fortunately, we can reasonably decide the issue of case (b) soon.

3.4.0.4 Distributing statically combined binaries require the permission of reverse engineering: Similar to Java, in C/C++ – the prototype of those languages, which are compiled as machine specific code – a C/C++ Library is also explicitly made known to the work that want to use it, namely by some include statements. These include statements denote the header files offered by and distributed with the Library. They contain the declarations of those elements which the Library wants to publish. Or briefly worded: the Library contains the definitions in form of code, the header files the corresponding declarations.

The C/C++ code following such include statements can refer to the definitions offered by the Library by using the declarations anounced by the header files. So, again, – from a strict viewpoint – the code of such a C/C++ work using the Library indeed contains portions of the library, even if these portions are only identifying names or data structures published by the header files.

Beyond that conceptual relation, the C/C++ development process finally compiles the work using the library as an object file containing machine specific code. Just as the Java compilation, also this process does not replace the referring names by the referred code of the Library; it still preserves the denoting names. The resulting file, which has been compiled into machine specific code, but still contains the denoting identifiers, is also known as 'object code file'.

[47] This directly follows from the LGPL2-RefEng-Rule by Modus Ponens

[48] This follows from the LGPL2-RevEng-Rule by Modus Ponens.

The C/C++ compilation process is (mostly) managed by a make file, which is executed by the make command[49]. This development tool calls the compiler for each source file, makes known the directories which contain the compiled target object files, and finally calls the linker. The linker recursively scans the compiled object files and replaces each embedded identifier by a truely executable jump command into that set of Library commands which are denoted by the identifier and which shall be executed as part of the work using the Library. So, only just at the end, the linker collects all necessary commands of all 'joined' object files and Libraries and produces the really executable work.

But – notwithstanding the above – the linker can either be called as integrated step of the developement process itself. Or the linker can seperatedly be called, especially on another machine: In the first case, the development process generates a *statically linked executable* which already contains all necessary portions of all used Libraries. In the second case, the development process generates a *dynamically linkable program* by collecting the (set of) still unlinked object code file(s) as a distributable package. Thus, if you distribute a statically linked executable, it definitely contains 'portions' of the library; if you distribute a dynamically linkable program you have to decide whether the embedded identifying names of a Library have indeed to be interpreted as portions of the Library.

Unfortunately, we still have to consider a little complication, based on the nature of the a C/C++ development process: In contrary to the Java development environment, a C/C++ development process inherently uses a preprocessor engine. This engine takes the header files delivered by the Library, verifies the syntactically correct use of the Library and can indeed replace some tokens of the work using the Library by commands and/or lines from the Library. This technique is known as *inline functions* or *macros*. They have been invented for those cases where expanding the stack of commands during the compilation by a real function call is more expensive than writing the embedded commands of the function more than one time into the whole code. Hence, in the C/C++ development process the compiled object files can indeed contain more than only the referring names which denote portions of the Library: beside the denoting identifiers, they can also already contain real, functionally relevant portions of the Library.

Thus, – again and similar to Java compilation – we may conclude, that with respect to C/C++ programming you (a) have to permit reverse engineering, if you distribute your work using the Library together with the Library as a statically linked program[50] and that (b) in all other cases your obligation to permit reverse engineering depends on the interpretation whether the used identifiers or dissolved

[49] Sometimes there additionally exist a complete meta environment which generates such make files. The GNU build system for example offers a complex set of configure scripts and make file templates (cf. http://en.wikipedia.org/wiki/GNU_build_system, wp.).

[50] This follows from the LGPL2-RevEng-Rule by Modus Ponens.

inline functions and macros, which have been declared by the Library and which therefore have automatically and standard conformably been embedded into an object file, are indeed portions of the Library.

Obviously, it is time to answer this crucial question:

3.4.0.5 Distributing dynamically combinable bytecode and linkable object code does not require the permission of reverse engineering: Of course, there is only one instance, that can answer the question, whether indentifiers and dissolved inline-functions or macros, which are – according to the development standard – embedded into a work using the Library, indeed are portions of the Library. This instance is the LGPL-v2 itself. And – fortunately – this license supports us in a very clear way to answer this question, even if not by its §6 which deals with the reverse engineering, but by its §5:

The LGPL simply specifies that "linking a 'work that uses the Library' with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library) [...]" and that "the executable is therefore covered by this License"[51]. Additionally, it talks about compiled, but still unlinked "object files", which therefore are not executables . Such an unexecutable "object file" – for example that of the "work using the Library" –, which "[...] uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length)" shall practically not be covered by the license of the Library, because "[...] the use of the object file is unrestricted regardless of whether it is legally a derivative work"[52] - as long as it does not exceed the given limits.

Obviously, the answer of the LGPL to our question is this: (a) yes, such object files containing names and snippets offered by the used Library, could contain portions of the Library. But it is not necessary to clarify the details, because (b) – up to a specific limit of sizes – these kind of 'little' portions being embedded into the object file by the standard compilation processes do not evoke any requirements: they especially do not evoke the obligation to allow reverse engineering. In other words: These little portions of a Library which are embedded by the standard development process and which do not contain more than the specified size of code may be regarded as another type of portions compared to the normal, real portions which indeed evoke the obligation to allow reverse engineering. From the viewpoint of the LGPL, they are *pseudo portions* of the Library, because they do not restrict the containg object file in any respect.

So, from the LGPL-RevEng-Rule we can now indirectly conclude, that distributing dynamically linkable or combinable bytecode or object code files which contain

[51] cf. *Open Source Initiative*: The LGPL-2.1 License (OSI), 1999, wp. §5.
[52] cf. id., ibid.

"only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length)" being delivered by a Library does not require to allow reverse engineering[53].

Unfortunately, there might be a practical objection which seems to disturb our simple result: For applying this rule correctly, we apparently have to assure that a compiled work that uses the Library but is still not *joined to it, indeed has only been expanded by "small macros or small inline functions (ten lines or less in length)". Thus, seemingly, we have to study all header files of all used Libraries in detail, if we want to compliantly distribute a work using a Library without permitting reverse engineering. This could be a lot of work – up to a bulk which practically can not be managed.

Fortunately, there is a simple solution for this challenge, a rule of thumb, based on the principle "trust the upstream"[54]:

The Library developers of course publish the header files or the public members and functions of the classes in exactly that form they want these elements to be used. And they want their Library to be used as an LGPL library, otherwise they would have chosen another License. So, they wish that improvements of the Libraries shall be made accessible as well, but that the works using the Library shall not necessarily be published in form of source code[55]. Thus, as long as we use a Library exactly in that form, the original authors have published, as long as we load down the Library from the official repository, and as long as we do not modify the intended interfaces defined and published by the original header and class files, we may justifiably assume that we are using the Libraries just as their copyright owners want them to be used. And thus, – in other words: as long as we trust the upstream – we might assume that the header and class files of our Libraries fit the restrictions of the LGPL-v2.

3.4.0.6 LGPL-v2 compliance with or without permitting reverse engineering:

Now, we have reached our target. Our last clarification can directly be applied to the both open cases: to the case of distributing Java bytecode as well, as to the case of distribution C/C++ object code. We now know, that the LGPL-v2 wishes, that not all portions of a Library covered by a work using the Library, trigger

[53] From the decision not to allow reverse engineering follows by Modus Tollens applied to the LGPL2-RevEng-Rule, that the distribution of the work using the Library must not contain real portions of the Library. From LGPL-v2-§5 and the limit of the standard proccesses follows that here the work using the Library does not contain normal, real portions. So, we know, that this case is not covered by the LGPL2-RevEng-Rule and thus we are allowed, to distribute a work using the Library without allowing its reverse engineering.

[54] On the ELLW 2013, we were told about this principle for the first time. We do not know, whether Armijn Hemel invented it. But we can respectfully affirm that he has persuasively explained the spirit and purpose of the principle "trust the upstream".

[55] The meaning of the weak copyleft.

the permission of reverse engineering. And we now know that the limits – given by the LGPL-v2-§5 – up to which such pseudo portions indeed do not trigger the obligation to permit reverse engineering, are respected, if we use *'upstream approved'* C/C++ and Java libraries in standard development environments. Thus, we indeed finally may conclude, that the LGPL-RevEng-Sentence

> *"[...] you may [...] combine* ***or*** *link a 'work that uses the Library' with the Library to produce a work containing portions of the Library* ***and*** *distribute that work under terms of your choice,* ***provided that*** *the terms permit modification of the work for the customer's own use* ***and*** reverse engineering *for debugging such modifications."*[56]

means 'nothing else' than

- *With respect to a LGPL-v2 licensed Library, you are not required to allow reverse engineering, if you [A] develop your work using the Library, on the base of a standard version of the Library containing the interfaces as the original developers have designed it, if you [B] compile your work using this Library, as a discret (set of) dynamically linkable or combinable file(s), if you [C] use only the standard compilation methods which preserve the upstream approved interfaces*[57]*, and if you [D] distribute the produced unlinked object code or bytecode files before they are linked as an executable.*
- *In all other cases of distributing a work using such a Library, you are required to allow reverse engineering of the work using this Library – especially, ...*
  - *if you distribute the work using the Library and the Library together as a statically linked program or as an integrated package containing both parts, the work using the library and the Library itself*[58]*.*
  - *if you distribute a work containing manually copied portions of the Library.*

3.5 Final Securing

So far, we have done a lot of work: At first, we unfolded and dissolved some stylisch condensed formulations of the original LGPL2-RevEng-Sentence by their linguistically explicit version. At second, we explicated the logical structure of the sentence. At third, we empirically carved out the real meaning of the sentence. And finally we mapped the triggering part of that rule to some verifiable facts.

---

56) cf. *Open Source Initiative*: The LGPL-2.1 License (OSI), 1999, wp., §6, emphasis KR..

57) and which therefore do not to exceed the LGPL-v2 limits!

58) This holds also if you distribute a script language based program or package, notwithstanding the fact, that one does not need the permission of reverse engineering to understand script language based applications.

Indeed, a lot of work for understanding only one sentence correctly[59]. So, it is a good securing to verify that the derived result fits the spirit and the goals of the LGPL-v2 perfectly.

For that purpose, let us fist discuss a little (semi-) official article – written by David Turner and published by the FSF[60] – which deals with the compliant use of LGPL licensed Java libraries. Turner refers to the "FSF's position" which - as he says - "[...] has remained constant throughout":

> *"[...] the LGPL works as intended with all known programming languages, including Java. Applications which link to LGPL libraries need not be released under the LGPL. Applications need only follow the requirements in section 6 of the LGPL: allow new versions of the library to be linked with the application; and allow reverse engineering to debug this."*[61]

Then he describes, that Java libraries are "[...] distributed as a separate JAR (Java Archive) file" and that applications "[...] use Java's 'import' functionality to access classes from these libraries". Moreover, he also explains, that the process of compilation "creates" and integrates "links" into the compiled application which let become the application a "derivative work" of the library. Finally he states, that not only the LGPL permits to distribute such derivative works, but that "[...] it is easy to comply with the LGPL" if one indeed wants to "[...] distribute a Java application that imports LGPL libraries": "Your application's license needs to allow users to modify the library, and reverse engineer your code to debug these modifications."[62]

So, we might state, that even this semi-official article argues very similarly to us. There is only one little phrase in this text which differs a little: Summarizing the "section 6 of the LGPL" by the statement "*[...] allow new versions of the library to be linked with the application; and allow reverse engineering to debug this*" does not consider that the first sentence of the section 6 of the LGPL contains a

---

59) Here, some readers might ask why the original authors have encapsulated their clear ideas in such a sophisticate sentence. Here are two answers: First, this question is practically irrelevant: The authors of the LGPL-v2 did, what they have done. And many developers have already licensed their works under the terms of the LGPL-v2. Thus, we simply have to live with the results – just until the last software being published under the terms of the LGPL-v2 is relicensed by a better version. Probably this won't happen during our life time. Secondly, we appreciate the foresight of the LGPL-v2 authors. They wrote a license which have successfully worked for more than twenty years. They chosed a formulation which had also to cover 'uninvented' techniques. So, it is not so surprizing, that we – today – have to do a lot of work to understand all details the original authors want to be understood.

60) cf. *Turner, David*: The LGPL and Java; 2004 ⟨URL: http://www.gnu.org/licenses/lgpl-java.en.html⟩ – reference download: 2015-02-09, wp..

61) cf. id., l.c., wp.

62) cf. id., l.c., wp..

complex condition. The *LGPL2-RefEng-Sentence* means – as we could prove – that *one may distribute (a)* ***work containing portions of the Library*** *only if one's license permit reverse engineering for debugging modifications*[63]. But – as we could also show – for determining wether an application really contains portions of the Library, one has additionally to consider the limits defined by section 5 of the LGPL[64]: the application's license needs to allow to reverse engineer the application only if it contains more elements of the Library than §5 of the LGPL-v2 has specified as limit.

That our analysis fits the spirit of the LGPL, can also be shown by considering the LGPL directly:

The LGPL-v2 clearly describes its goals. It wants to enable the community to let an LGPL Library "[...] become a de-facto standard". And the LGPL knows, that "to achieve this [goal], non-free programs must be allowed to use the library", because the "[...] permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software". But the LGPL also asserts in this context, that "although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library"[65].

So – as a last check of our derivation – we can analyze, whether our derived result violates this goal. If it does, then we probably made a tremendous fault; if not, then we are allowed to trust in the consistence our analysis:

If you receive a work using the Library in form of a discret (set of) dynamically linkable or combinable file(s) and if – hence – your provider assumed that the files he delivers will be linked on your target machine which – therefore – has to provide a linker and the the necessary dynamically linkable Libraries, than you systematically have the freedom to replace the dynamically linked Libraries by their updated versions[66]. And as long as the newer versions of the Libraries preserve the defined and declared interfaces, you can do that successfully. That's, what the LGPL-v2 wants to ensure.

In all other cases, you must have the permission of reverse engineering or you have a direct access to the source code. So, you can use the corresponding tools and techniques to replace the embedded version of the Library by a newer version; especially if you have received a statically linked package. Hence, also the second

63) → p. 17
64) → p. 23
65) cf. *Open Source Initiative*: The LGPL-2.1 License (OSI), 1999, wp., preamble, emphasis KR.
66) In GNU/Linux – for example – you must (only) copy the dynamically linkable new version of the Library into the lib/-directory and replace the existing link by a version pointing to the newer version. Sometimes you should additionally verify the ld.so.conf files and call ldconfig tool.

part of our interpretation respects the spirit of the LGPL-v2.

So, finally we can say, everything is fine: The LGPL2-RevEng-Rule – together with the meaning of being a portion of a Library – does not only verifiably exeplicate the meaning of the LGPL2-RevEng-Sentence, but also fits the spirit and the purpose of the LGPL-v2 as it has been announced by its preamble.

4 Reverse Engineering in the LGPL-v3

Based on our experiences how to successfully carve out the meaning of license text, we can shorten the way to understand the one LGPL3-RevEng-Sentence referring to *reverse engineering*:

> *"You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following [...]"*[67]

Reusing our method of disambiguation, we first can exemplify the meaning of the LGPL3-RevEng-Sentence by the following text:

```
( [Θ :]
  ( You compliantly distribute a Combined Work
    under terms of your choice
    (   (that together effectively, do not restrict modification of
        the portions of the Library contained in the Combined Work)
    AND
        (that together effectively, do not restrict reverse
        engineering for debugging modifications of the portions
        of the Library contained in the Combined Work)
  )  )
  IF
  [Ω :]
  ( you also do each of the following [...])
)
```

But now, a simply executed logical serialization let us running into a problem:

If we serialized *(Θ IF Ω)* as *(Ω → Θ)*, then from not respecting Θ would follow by Modus Tollens, that we are not allowed to realize Ω – in other words: that we

[67] cf. *Open Source Initiative*: The LGPL-3.0 License (OSI), 2007, wp., §4. The ellipsis at the end of the sentence denotes a set of tasks which we do not listen here for saving recourses, but which have to be considered as an integrated part of this sentence.

may not do even one of the single tasks covered by the ellipsis – which is a silly result.

If we serialized *(Θ IF Ω)* as *(Θ → Ω)* then from doing Θ would successfully follow by Modus Ponens that we also have to do Ω. And from not respecting Ω would successfully follow by Modus Tollens, that we must not do Θ. But unfortunately, we can respect this second interdiction also *by distributing a Combined Work under terms* that restrict modifications and/or reverse engineering (instead of not restricting these techniques) – which, again, is a silly result.

Obviously, a simple serialization based on a intutively unclear reading fails. In fact, the LGPL3-RevEng-Sentence must have a more sophisticated underlying structure. It must be logically serialized in a form, that integrates the requirements, not to restrict modifications and reverse enigneering, as really triggable conditions. Thus, the meaning of the sentence can logically be explicated as the *LGPL3-RevEng-Rule*:

```
( [Σ :]
  ( You compliantly distribute a Combined Work
    under terms of your choice
  )
  →
  (    [Γ :]
     ( the terms of your choice together effectively do
       not restrict modification of the portions of the
       Library contained in the Combined Work)
     ∧ [Δ :]
     ( the terms of your choice together effectively, do
       not restrict reverse engineering for debugging
       modifications of the portions of the Library
       contained in the Combined Work)
     ∧ [Ω :]
     ( you also do each of the following [...])
) )
```

This LGPL3-RevEng-Rule indeed successfully regulates how to compliantly distribute a Combined Work by telling us,

- that we have to respect Γ, Δ **and** all single parts of Ω, if we distribute a Combined Work compliantly[68].

[68] follows by Modus Ponens. Thus, in this case especially our terms "[...] together effectively **[must] not restrict reverse engineering** for debugging modifications of the portions of the Library contained in the Combined Work".

- that we do not distribute a Combined Work compliantly, if we do not respect one of the requirements Γ, Δ or one of the single parts of Ω[69].

Now, we can directly see, that the LGPLv3 does not enforce us, not to obstruct reverse engineering in all respects! The required reverse engineering is limited to the purpose of supporting the debugging of modifications and focused to the Combined Work containing portions of the Library. In other words: our terms may obstruct other purposes of reverse engineering or may restrict reverse engineering of other forms of our work which which can not be specified as Combined Work or do not contain portions of the Library. Thus, the first crucial question is, what the LGPL-v3 means if it talks about a "Combined Work". The second question is, what the LGPL-v3 specifies as a portion of the Library.

Again, fortunately, the LGPL-v3 answers clearly: "A 'Combined Work' is a work produced by combining or linking an Application with the Library"[70]. From our LGPL-v2 analysis we know the ways how works that uses a Library can technically be linked or combined with the Library:

- Copying code from the Library into the work using the Library[71] causes that the application respectively the work using the Library indeed contains portions of the Library[72].
- Combining script language based applications and Libraries may evoke that the resulting application contains portions of the Library. But the details can be neglected with respect to the reverse engineering, because script code is distributed as it has been developed and can therefore directly be understood[73].
- Combining java classes and libraries as integrated quasi statically linked packages causes, that the resulting package already contains all functionally necessary code of the Library[74].
- Compiling java classes without combining them with the referred Library classes causes, that the compiled classes at least contain identifiers having been declared by the Library[75].
- Combiling C/C++ files or classes and linking them with the referred Libaries

[69] follows by Modus Tollens. Thus, especially we are not distributing a Combined Work compliantly, if our terms "[...] together effectively **do restrict reverse engineering** for debugging modifications of the portions of the Library contained in the Combined Work".
[70] cf. *Open Source Initiative*: The LGPL-3.0 License (OSI), 2007, wp., §0.
[71] The LGPL-v3 designates the work using the Library as "Application" and defines that it "[...] makes use of an interface provided by the Library [...]" (cf. id., ibid.).
[72] → p. 18
[73] → p. 20
[74] → p. 20
[75] → p. 23

statically causes, that the resulting executable indeed contains all functional relevant code of all used Libraries[76].

- Combiling C/C++ files or classes without linking them to the referred Libaries causes, that the resulting object file can dynamically be linked on another machine and contains identifiers offered by the Library and sometimes some functional code injected by dissolving some inline functions or macros[77].

So – overall – the situation is this: The LGPL3-RevEng-Rule tells us that we have to allow reverse engineering of the portions of the Library contained in the Combined Work. The LGPL3 additionally specifies, that a Combined Work is simply the result of technically combining the work using the Library (the application) and the Library. Finally the praxis tells us, that (a) combining both components statically indeed causes that the resulting Combined Work contains portions of the Library[78], and that (b) we – in case of preparing the both parts as dynamically combinable components – still have to clarify whether the resulting work already contains portions of the Library.

Just as the LGPL-v2, the LGPL-v3 supports us to answer this question by its §3 whose linguistic conjunctions we thoroughly have to consider:

> *The object code form of an Application may incorporate material from a header file that is part of the Library.* ***You may convey*** *such object code under terms of your choice,* provided that, *[* ***if*** *the incorporated material is* ***not*** *limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length)],* ***you do both*** *of the following:* ***a)*** *Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.* ***b)*** *Accompany the object code with a copy of the GNU GPL and this license document]*[79].

The first sentence of this paragraph tells us that he is dedicated to object files which are compiled and not linked to the used Library, but which nevertheless can contain portions of the Library. The second sentence regulates the distribution of such object files and can be logically serialized:

```
( [Λ :]
  ( You compliantly distribute object code [incorporating
```

[76] → p. 21
[77] → p. 23
[78] So, it is triggering the LGPL3-RevEng-Rule.
[79] cf. *Open Source Initiative*: The LGPL-3.0 License (OSI), 2007, wp., §3; emphasis and additional braces KR..

```
    material from the Library] under terms of your choice )
  →
  [Ξ :]
  ( [ω :]
    ( the incorporated material is not limited to numerical
      parameters, data structure layouts and accessors, or
      small macros, inline functions and templates
      [ten or fewer lines in length] )
    →
    ( [α :] ( you do [a] ...] )
    ∧ [β :] ( you do [b] ...] )
) ) )
```

We see, that this LGPL3-sentence concerning the distribution of object files contains a main rule $(\Lambda \rightarrow \Xi)$ and that the conclusion Ξ itself has the form of an embedded sub rule $(\omega \rightarrow (\alpha \wedge \beta))$.

Firstly, the main rule enforces us to respect the sub rule if we want to distribute the object code compliantly[80]. Secondly, the main rule tells us that we do not distribute the object code compliantly if we do not respect the sub rule [81].

We have two ways to respect the sub rule, and one way not to respect it:

- If the object code contains more and/or larger elements of the Library than the limit specifies, then **we do respect the sub rule**, if we do α and β[82].
- If the object code contains elements of the Library at most up to specified limits, then **we do respect the sub rule** without having to do some additionally tasks[83]
- But if the object code contains more and/or larger elements of the Library than the limit specifies **and** if we do not do α *or* β, then **we do not respect the sub rule**[84].

Thus, – at the end and based on the additional object code specification and the known empirical background knowledge concerning the software programming – the LGPL3-RevEng-Rule delivers the same result as the LGPL2-RevEng-Rule[85]:

- *With respect to a LGPL-v3 licensed Library, you are not required to allow reverse engineering, if you [A] develop your work using the Library, on the*

[80] follows by Modus Ponens to $(\Lambda \rightarrow \Xi)$.
[81] follows by Modus Ponens to $(\Lambda \rightarrow \Xi)$.
[82] follows by Modus Ponens to $(\omega \rightarrow (\alpha \wedge \beta))$.
[83] follows by definition of an implication: if the premise of this sub rule is false, the sub rule is as whole is true and hence respected.
[84] follows from definition of an implication: if the premise is true and the conclusion is false, the the implication as whole is false, as well.
[85] → 24

base of a standard version of the Library containing the interfaces as the original developers have designed it, if you [B] compile your work using this Library, as a discret (set of) dynamically linkable or combinable file(s), if you [C] use only the standard compilation methods which preserve the upstream approved interfaces[86]*, and if you [D] distribute the produced unlinked object code or bytecode files before they are linked as an executable.*

- *In all other cases of distributing a work using such a Library, you are required to allow reverse engineering of the work using this Library – especially, ...*
  - *if you distribute the work using the Library and the Library together as a statically linked program or as an integrated package containing both parts, the work using the library and the Library itself*[87]*.*
  - *if you distribute a work containing manually copied portions of the Library.*

5 Reverse Engineering in the other Open Source Licenses

The rest of our way is simple: First, we can ascertain, that none of the other open source licenses we consider[88], contain the phrase 'reverse engineering'. Moreover, they even do not contain one of the single words[89]. So, we may infer, that these most important other open source licenses could at most indirectly require the permission of reverse engineering. Second, we know already that distributing script code let the allowance to reverse engineer, become irrelevant: script code can directly be read and understood, if one knows the script language[90]. Third, from the definition of strong copleft we may derive, that distributing software licensed under a strong copyleft license let the permission of reverse engineering become unimportant, because the source code of the work using the libraries licensed under a copleft license, must also be made accessible[91].

So – overally – we may conclude, that we have only to consider those cases, where

86) and which therefore do not to exceed the LGPL-v3 limits

87) This holds also if you distribute a script language based program or package, notwithstanding the fact, that one does not need the permission of reverse engineering to understand script language based applications

88) → p. 6

89) One can verify this negative statement by (a) loading down all licenses from the OSI homepage (http://opensource.org/licenses/alphabetical) and by (b) executing the command `grep -i "engineering" *` respectively `grep -i "reverse" *` in the directory into which the license files have been stored: grep will find the words *reverse* and *engineering* only in the texts of the LGPLs.

90) → p. 20

91) cf. *Stallman, Richard M.*: What is Copyleft? originally written in 1996; in: *Joshua Gay*, editor: Free Software, Free Society: Selected Essays of Richard M. Stallman; Boston, MA USA: GNU Press, 2002, ISBN 1–882114–98–1, wp.

a piece of software is distributed in form of binaries or bytecode, which uses libraries licensed under permissive open source licenses or under weak copyleft licenses.

From the definition of being a permissive license or a weak copyleft license we know already that the licenses of the open source components do not directly influence the permission or interdiction to use the overarching work which uses the open source software components[92].

So, if we distribute such a work in form of dynamically linkable, but still not linked binaries or bytecode files, then there is no way to reasonably derive that the work using the components, may be reverse engineered: The permissive or weak copyleft open source licenses mainly concern the open source components, not the work using the components. On the one side, these licenses indeed require that we add the license texts and the copyright lines of all the open source components our work wants to use, to the distributed package containing our work. And the lisenses prohibit to modify the licensing assertions being integrated into the open source components our work wants to use[93]. But – on the other side and in accordance to the permissive or weak-copyleft licenses – the freedom to use, to study, to modify, or to distribute the software, which is established by these open source licenses, concerns only the open source components themselves, not the work using the open source components. So, as long as these components still are not linked to or combined with the using work in accordance to the standard compilation and computation methods, they can indeed be studied or modified without the need to study or modify the work which uses these components[94].

---

92) cf. *Reincke, Karsten*, *Greg Sharpe*, a. *contributors*: Open Source License Compendium. How to Achieve Open Source License Compliance; 2015 ⟨URL: http://www.oslic.org/releases/oslic.pdf⟩ – reference download: 2015-01-20, pp. 20ff..

93) These requirements are part of all the open source licenses we consider here. For details cf. id., l.c., pp. chapter 6.

94) The only way to infer that the licenses of the components operates also on the using work, is to argue that the using work must at least contain elements (identifiers etc.) of the interfaces declared (but not defined) by the libraries and that therefore at least these elements may be investigated or modified. This challenge is explicitly addressed by the LGPL[95]. Fortunately, it is a general feature of software libraries that they must and shall be used in accordance to the interfaces, the developers of the libraries have designed to make their libraries practically usable. So, if the licenses – in contrary to the LGPLs – do not explicitly address the issue of implicitly included portions of the library in case of unlinked binaries or bytecode files which have been compiled in accordance to the standard methods and which therefore use open source software by reffering to their standard interfaces, then one has to infer from the nature of computation, that the developers have implictly allowed without any requirements such an integration of declared, but not defined interface elements, because they have designed the interface as they did and because they have licensed their work as they did. If they had not wished to use these elements without any requirements, hey had designed another interface. And if they had wished to incorporate any copyleft effect or permission of reverse engineering, then they would have selected another license. But again: this conclusion holds only for the standard methods to use a software library.

On the other side, if we compliantly distribute the work using the components, as a statically linked binary or bytecode file – which therefore already contains all the necessary components[96] and can directly be executed –, then we are also obliged to add all the open source license texts and all the copyright lines to our package, and we are not allowed to modify one of the licensing assertions integrated into the original open source components[97]. Thus, one might conclude, that the freedom to use and to modify the open source components themselves, survive if we distribute software statically linked to or combined with the open source components. So, the receiver of the statically linked work probably is allowed to modify the embedded open source components - even if he had to edit the binary or bytecode files. Methods to develop binary files reversely, are known as reverse engineering. Hence, if we distribute a statically linked work using open source licensed components, we have at least to fear that our receivers indirectly have also got the permission to reverse engineer our complete product. And we have to fear so even if the statically linked libraries are licensed under any permissive or weak copleft license.

So, again, we can summarize the result in the following form:

- *With respect to a Library licensed under any permissive or weak copyleft license, you are not required to allow reverse engineering, if you [A] develop your work using the Library, on the base of a standard version of the Library containing the interfaces as the original developers have designed it, if you [B] compile your work using this Library, as a discret (set of) dynamically linkable or combinable file(s), if you [C] use only the standard compilation methods which preserve the upstream approved interfaces, and if you [D] distribute the produced unlinked object code or bytecode files before they are linked as an executable.*
- *In all other cases of distributing a work using such a Library, you have at least to fear that you are implictly allowing reverse engineering of the work using this Library – especially, ...*
  - *if you distribute the work using the Library and the Library together as a statically linked program or as an integrated package containing both parts, the work using the library and the Library itself*[98].
  - *if you distribute a work containing manually copied portions of the Library.*

[96] instead of only the declared interface elements!

[97] cf. *Reincke, Sharpe, a. other contributors*: OSLiC, 2015, pp. chapter 6..

[98] This holds also if you distribute a script language based program or package, notwithstanding the fact, that one does not need the permission of reverse engineering to understand script language based applications

6 Reverse Engineering in Open Source Licenses: Summary

So, finally we can compile all our results into one single result:

- *With respect to any open source Library*[99]*, you are not required to allow reverse engineering, if you [A] develop your work using the Library, on the base of a standard version of the Library containing the interfaces as the original developers have designed it, if you [B] compile your work using this Library, as a discret (set of) dynamically linkable or combinable file(s), if you [C] use only the standard compilation methods which preserve the upstream approved interfaces*[100]*, and if you [D] distribute the produced unlinked object code or bytecode files before they are linked as an executable.*
- *In all other cases of distributing your work using such a Library, you are probably required to allow reverse engineering of your work. By all means, you have at least to fear that you are implictly allowing reverse engineering of your work using such a Library – especially, ...*
  - *if you distribute the work using the Library and the Library together as a statically linked program or as an integrated package containing both parts, the work using the library and the Library itself*[101]*.*
  - *if you distribute a work containing manually copied portions of the Library.*

And, so, we can reformulate our result as a slightly modified "rule of thumbs" originally offered by an open source expert who analyzed the problem of protecting your own work from an other viewport:

- "DO NOT statically link [or combine] [open source] code if you wish to keep your program proprietary [and if you want to protect it against reverse engineering]"[102].
- "DO dynamically link to [any open source code, not only to] LGPL code"[103].

q.e.d

[99] → p. 6

[100] and which therefore do not to exceed limits, prescribed by the owners of the Library

[101] This holds also if you distribute a script language based program or package, notwithstanding the fact, that one does not need the permission of reverse engineering to understand script language based applications

[102] cf. *Ilardi*: Common OSS License Problems, 2010, pp. 6; bracketed text KR..

[103] cf. id., ibid.

7 Disclaimer

This article is thoroughly developed. Finally – and preferably with the help of the open source community –, it shall deliver reliable information. But nevertheless, it can not offer more than the opinion(s) of its authors and contributors. It is only one voice of the chorus discussing the open source licenses. For protecting the authors and contributors from charges and claims of indemnification we adopt the lightly modified GPL3 disclaimer:

THERE IS NO WARRANTY FOR THIS ATRTICLE, TO THE EXTENT PERMITTED BY APPLICABLE LAW. THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE TEXT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE OSLiC IS WITH YOU. SHOULD THE OSLiC PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE OSLiC AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS ARTICLE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE OSLiC TO COOPERATE WITH ANY OTHER TOOLS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

8 License

Deutsche Telekom AG into your preface. For normal citations please use the scientific standard.

[LaTeX form derived from myCsrf (= 'mind your Scholar Research Framework') ©K. Reincke CC BY 3.0 http://mycsrf.fodina.de/)]

References

copyleft.org: What is copyleft.org; n.l, 2014, FreeWeb/HTML ⟨URL: http://copyleft.org/⟩ – reference download: 2014-12-15

Ilardi, Terry J.: Common OSS License Problems; n.l, 2010, FreeWeb/PDF ⟨URL: http://www2.aipla.org/html/spring/2010/papers/Ilardi_Paper.pdf⟩ – reference download: 2014-12-16

Jaeger, Till a. *Axel Metzger*: Open Source Software. Rechtliche Rahmenbedingungen der Freien Software; 3rd edition. München: Verlag C.H. Beck, 2011, Print

Kuhn, Bradley M. et al.: Copyleft and the GNU General Public License: A Comprehensive Tutorial and Guide; n.l, 2014, FreeWeb/PDF ⟨URL: http://copyleft.org/guide/comprehensive-gpl-guide.pdf⟩ – reference download: 2014-12-15

Moglen, Even a. *Mishi Choudhary*: Software Freedom Law Center Guide to GPL Compliance, 2nd Edition; 2014, FreeWeb/HTML ⟨URL: https://www.softwarefreedom.org/resources/2014/SFLC-Guide_to_GPL_Compliance_2d_ed.html⟩ – reference download: 2014-12-15

Open Source Initiative: GNU General Public License, version 2 (GPL-2.0). Version 2, June 1991; 1991 [n.y. of the html page itself], FreeWeb/HTML ⟨URL: http://opensource.org/licenses/GPL-2.0⟩ – reference download: 2013-02-05

Open Source Initiative: The GNU Lesser General Public License, version 2.1 (LGPL-2.1); 1999 [n.y. of the html page itself], FreeWeb/HTML ⟨URL: http://opensource.org/licenses/LGPL-2.1⟩ – reference download: 2013-03-06

Open Source Initiative: Apache License, Version 2.0; 2004 [n.y. of the page itself], FreeWeb/HTML ⟨URL: http://opensource.org/licenses/Apache-2.0⟩ – reference download: 2013-02-07

Open Source Initiative: Eclipse Public License, Version 1.0; 2005 [n.y. of the page itself], FreeWeb/HTML ⟨URL: http://opensource.org/licenses/EPL-1.0⟩ – reference download: 2013-02-20

Open Source Initiative: European Union Public License, version 1.1 (EUPL-1.1; 2007 [n.y. of the html page itself], FreeWeb/HTML ⟨URL: http://opensource.org/licenses/EUPL-1.1⟩ – reference download: 2013-03-04

Open Source Initiative: GNU Affero General Public License, Version 3 (AGPL-3.0); 2007 [n.y. of the html page itself], FreeWeb/HTML ⟨URL: http://opensource.org/licenses/AGPL-3.0⟩ – reference download: 2013-04-05

Open Source Initiative: GNU General Public License, version 3 (GPL-3.0); 2007 [n.y. of the html page itself], FreeWeb/HTML ⟨URL: http://opensource.org/licenses/GPL-3.0⟩ – reference download: 2013-03-05

Open Source Initiative: The GNU Lesser General Public License, version 3.0 (LGPL-3.0); 2007 [n.y. of the html page itself], FreeWeb/HTML ⟨URL: http://opensource.org/licenses/LGPL-3.0⟩ – reference download: 2013-03-06

Open Source Initiative: The BSD 2-Clause License; 2012 [n.y.], FreeWeb/HTML ⟨URL: http://www.opensource.org/licenses/BSD-2-Clause⟩ – reference download: 2012-07-03

Open Source Initiative: The BSD 3-Clause License; 2012 [n.y.], FreeWeb/HTML ⟨URL: http://www.opensource.org/licenses/BSD-3-Clause⟩ – reference download: 2012-07-04

References

Open Source Initiative: The MIT License; 2012 [n.y.], FreeWeb/HTML ⟨URL: http://opensource.org/licenses/mit-license.php⟩ – reference download: 2012-08-24

Open Source Initiative: Microsoft Public License (MS-PL); 2013 [n.y.], FreeWeb/HTML ⟨URL: http://opensource.org/licenses/MS-PL⟩ – reference download: 2013-02-26

Open Source Initiative: Mozilla Public License 2.0 (MPL-2.0); 2013 [n.y.], FreeWeb/HTML ⟨URL: http://opensource.org/licenses/MPL-2.0⟩ – reference download: 2013-02-07

Open Source Initiative: The PHP License 3.0 (PHP-3.0); 2013 [n.y.], FreeWeb/HTML ⟨URL: http://opensource.org/licenses/PHP-3.0⟩ – reference download: 2013-02-27

Open Source Initiative: The PostgreSQL Licence (PostgreSQL); 2013 [n.y.], FreeWeb/HTML ⟨URL: http://opensource.org/licenses/PostgreSQL⟩ – reference download: 2013-02-27

Reincke, Karsten, *Greg Sharpe*, a. *contributors*: Open Source License Compendium. How to Achieve Open Source License Compliance; 2015, FreeWeb/PDF ⟨URL: http://www.oslic.org/releases/oslic.pdf⟩ – reference download: 2015-01-20

Rosen, Lawrence: Open Source Licensing. Software Freedom and Intellectual Property Law; Upper Saddle River, New Jersey: Prentice Hall PTr, 2005, ISBN 0–13–148787–6

Stallman, Richard M.: What is Copyleft? originally written in 1996; in: *Joshua Gay, editor:* Free Software, Free Society: Selected Essays of Richard M. Stallman; Boston, MA USA: GNU Press, 2002, ISBN 1–882114–98–1, pp. 89–90, Print

Turner, David: The LGPL and Java; 2004, FreeWeb/HTML ⟨URL: http://www.gnu.org/licenses/lgpl-java.en.html⟩ – reference download: 2015-02-09