THE LINUX FOUNDATION | Research

WebAssembly (Wasm) for Legal Professionals

Exploring Current Parameters in License Compliance

December 2022

By Armijn Hemel, MSc, *Tjaldur Software Governance Solutions*

Contents

Why this document?

A technology that is currently gaining quite a bit of traction is WebAssembly (Wasm). Documentation for WebAssembly on the Internet primarily targets developers and focuses on how to use it or develop for it. What is lacking is documentation that looks at WebAssembly from an open source license compliance perspective. This document is an attempt to fill that void.

WebAssembly is not unique on a technical level (most of the techniques have been used in the past one way or another), but on a governance level, it is a technology that can fundamentally change the speed of the web. Unlike in the past, the technology is a collaborative effort by developers of all major browser engines, and its development is a part of W3C.

Because it is technologically similar to existing technologies, license compliance processes that other open source contexts use do not need to change with WebAssembly. Because the delivery mechanism for WebAssembly code is primarily over the network, where you look for compliance information may change.

This document is not a technical document. It is not a programming tutorial, and the simplification of and glossing over many technical details avoid complicating the goal of this document. Where necessary, (technical) documentation containing more information is linked.

This document is also not a legal document, and the reader should not draw any legal conclusions from this content. The purpose of this document is to function as a starting point for a discussion about what open source license compliance for WebAssembly could look like and its potential for implementation in several scenarios. A goal is to highlight potential license compliance pitfalls and help establish a common understanding of the facts, which may present potential legal issues.

WebAssembly is still evolving, so this document will likely be outdated at some point. Just because this document does not describe a certain scenario does not mean that it is, therefore, safe to use or that there are no potential license compliance issues.

What is WebAssembly?

WebAssembly is a recent technology, primarily intended for creating and deploying high-performance web applications in situations where JavaScript cannot give that performance, although it is not limited to this, and there are other use cases[1] as well. WebAssembly programs are downloaded from a server, and a virtual machine that is typically running inside a web browser executes the programs. To improve security, the virtual machine is sandboxed—code running inside the sandbox cannot access code, data, or resources outside of the sandbox unless it is explicitly allowed to access these.

One difference with JavaScript is that WebAssembly programs are downloaded as compiled programs, which can be directly executed by the virtual machine inside the browser without needing to be parsed and interpreted. This stands in contrast with JavaScript programs that are downloaded in (typically minified[2]) source code form, which are then interpreted by the browser.

Although the primary design of WebAssembly keeps the web browser in mind, as the client, there are also standalone WebAssembly virtual machines, such as WAMR[3], Wasmer,[4] and others, allowing for the use of WebAssembly outside of the context of the web browser.

The WebAssembly website[5] describes WebAssembly as follows:

> "WebAssembly (abbreviated 'Wasm') is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications."

On the Mozilla WebAssembly website,[6] it says the following:

> "WebAssembly is a new type of code that can be run in modern web browsers—it is a low-level assembly-like language with a compact binary format that runs with near-native performance and provides languages such as C/C++, C#, and Rust with a compilation target so that they can run on the web. It is also designed to run alongside JavaScript, allowing both to work together."

The execution of downloaded compiled code in a sandbox environment inside a browser is not a new concept, and there are several historical examples, most notably Java applets and ActiveX. What makes WebAssembly stand out from these historical examples is that WebAssembly is an open standard that teams of all major browser engines are working on instead of a technology pushed by a single vendor or a technology that only works in a single browser or that only works well if the browser is running on a certain client operating system (which was, for example, the case with ActiveX). WebAssembly does not limit itself to a single programming language—programs written in multiple languages can be combined into WebAssembly binaries.

Even though the purpose of the original design of WebAssembly was to speed up operations where JavaScript doesn't suffice, it is expected that developers will come up with an entirely new class of applications or uses that the WebAssembly designers never thought of. Some people are working on embedding WebAssembly virtual machines as a shared library to make it possible to enhance every program with WebAssembly.

WebAssembly formats

The WebAssembly documentation mentions several formats:

1. Source code (possibly written in any of several programming languages)
2. A binary format produced by a compiler that is then loaded into and executed by the virtual machine
3. A textual representation of the binary format, similar to instructions of an assembly language

Source code & binary format

The source code is what programmers will typically write. They compile the source code into binary code using a compiler such as Emscripten[7], which can compile any LLVM-supported language into a WebAssemblybinary. Currently, the compilers support C/C++, C#, and Rust well. Support for other languages, such as Python[8], is in development.

The binary format of the program consists of object code instructions that the virtual machine inside the web browser executes. A compiler from source code written by a programmer typically generates it.

Textual representation

The textual representation of the binary format is typically not source code, but it is the textual representation of the binary instructions that the sandbox can execute. It is equivalent to the binary format, and there are tools that can convert the binary instructions into the textual representation and vice versa.

The textual representation is an assembler-like language with instructions called "S-expressions" manipulating a stack[9]. For instance, a small part of the textual representation of the binary code of an example that will be introduced later in this document looks like this:

```
(func (;6;) (type 2) (result i32)
    (local i32 i32 i32)
    i32.const 1078
    local.set 0
    i32.const 0
    local.set 1
    local.get 0
    local.get 1
    call 51
    drop
    i32.const 0
    local.set 2
    local.get 2
    return)
```

These instructions manipulate the state of the virtual machine.

The syntax is reminiscent of, for example, assembler code for an ARM processor or a MIPS processor. Data is put onto a stack, popped from it, and manipulated. With some imagination, WebAssembly instructions can be seen as an instruction set, with the WebAssembly virtual machine as the CPU[10].

It is possible to write small programs directly using these assembler-like instructions, but it is more likely that programmers will write code in a higher-level language such as Rust, C, or C++ (for efficiency, more expression power, and reusability). A programmer would then use a compiler to translate the high-level source code into binary code or its textual representation.

While in almost all cases the virtual machines will execute the binary code, most WebAssembly engines can load and interpret the instructions of the textual representation "just in time."

WebAssembly example

From a compliance standpoint, it is important to understand a little bit about how the compilation process works. Specifically, what happens with comments containing licensing information during compilation? Is information from a source code file included in the distributed file, and is the necessary compliance information, if any, included in the binary format after compilation?

Let's look at an example from the Mozilla Developer Network[11]. Assume that the necessary components and tools have been installed, such as Emscripten (a compiler for WebAssembly) and wabt (a tool to convert between the binary and textual representation of the compiled code), and that compilation is done on a Linux system. The example consists of a very simple C file, stored in a file called hello.c:

```
#include <stdio.h>
int main() {
      printf("Hello World\n");
}
```

The Emscripten compiler then compiles this as follows:

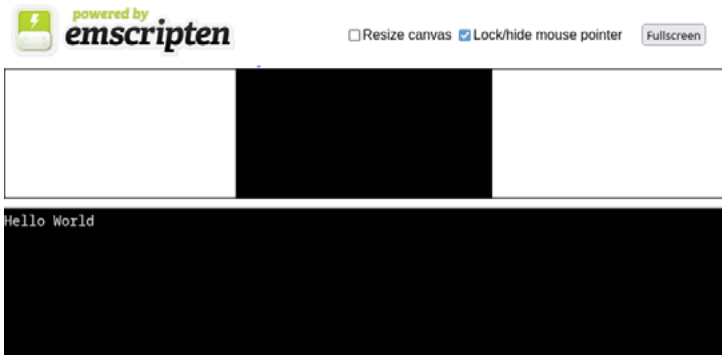
```
$ emcc hello.c -o hello.html
```

which will generate the following few files:

- hello.html
- hello.js
- hello.wasm

The Emscripten compiler generates the files hello.html and hello.js, which do not use any information from the C file except the name of the WebAssembly binary file. The file hello.wasm is a WebAssembly binary file compiled from the "hello.c" source code file.

During a web session where a user requests a resource from a web server, the web server sends the binary file to a client (typically the web browser, but could also be a headless client[12]), and it is executed in a sandbox environment running in the web browser. The web browser must support WebAssembly for this to work. The JavaScript code can be used to interact with the running WebAssembly binary and the .html webpage.



An interesting feature of WebAssembly is that the design of the instruction set allows for streaming instructions from the server to the client—the client doesn't need to wait for the server to finish sending all the binary code with instructions but can already start processing while receiving the rest of the code[13]. Therefore, it is possible to create a program that continuously sends WebAssembly code to a client as a continuous, never-ending stream of instructions. This code could include instructions compiled from open source code, which could add a unique twist to license compliance fact patterns.

The use of tools from the "wabt" package[14] can convert the binary file to a textual, assembly-like representation. The tool wasm2wat, for example, can dump the contents of the binary file into the textual representation of the WebAssembly instructions. The command

```
$ wasm2wat hello.asm
```

will output around 5,500 lines of assembly-like instructions for this simple Hello World example.

WebAssembly open source license compliance

Open source license compliance for WebAssembly is not very different from license compliance in other situations. In fact, it is pretty much the same, and existing licenses are applied to a new technical situation that is slightly different from current common patterns. WebAssembly does present new programming patterns that may be unique to many lawyers. But because it is so similar, there is also a lot of expertise available, and it is not necessary to reinvent the wheel. Instead, you can use best practices from, for example, the OpenChain Project[15].

Most open source licenses have clauses that are activated whenever distribution occurs. What you need to do when the code is distributed depends on the license. Some licenses require that the complete and corresponding source code (including license texts) is made available. For example, the GNU Public Licenses require either the distribution of source code together with the binary or that a written offer for the source code is given to the user. Other licenses may require that the license texts (and sometimes some other information) is included with the software (examples: MIT, 2-clause BSD, and Apache 2). The above categorization is a simplification, and some licenses will have varying requirements. It is therefore important to know what licenses of software you are using and what they require you to do.

The first question to ask is whether distribution takes place. In the standard WebAssembly use case, the answer to that would be "yes." Sending WebAssembly binary code from a web server to a client is typically a form of distribution. If you use any open source code, and there are clauses that are activated by distribution, then you need to fulfill the license obligations. Even with seemingly simple licenses, you may have other non-distribution-related license obligations as well.

For example, let's look at the MIT license[16], which is a popular license among website developers (a lot of JavaScript code is licensed under this license). It is also one of the shorter open source licenses; however, it contains license compliance requirements (as a condition to the license) when the binary software is distributed. You can find the following MIT license template on the SPDX website (emphasis added for this document):

> MIT License
>
> Copyright (c) <year> <copyright holders>
>
> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including, without limitation, the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
>
> **The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.**
>
> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT, OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The license basically says that every time a copy of the software is distributed in any form (source code or binary), the license text and copyright notice have to be distributed with the software as well. In the context of a WebAssembly codebase licensed under the MIT license, that would mean shipping the MIT license text and copyright notice with either the binary .wasm file or its textual representation.

So, how do you do this when most .wasm files are delivered behind the scenes for high optimization purposes to a web browser and intended to be invisible to the user? If the binary code contains the license information, and the user can easily access that license information, then you have met the condition of the license. The next question is as follows: Is the license information included with the binary when you compile the code?

As a test, the file "hello.c" from before was copied to "hello_license.c," and the above MIT license text was added as a comment at the top of "hello_license.c." The file with the license text is significantly larger than the original file. Whereas the normal source code file is a mere 65 bytes, the source code file with license information is 1217 bytes:

```
$ du -b *.c
65    hello.c
1217    hello_license.c
```

To verify what the compiler does with the license text at the top of the second file, the two C files were compiled into wasm files:

```
$ emcc hello.c -o hello.html
$ emcc hello_license.c -o hello_license.html
```

and compared for size and MD5 checksum:

```
$ du -b *.wasm
12394    hello_license.wasm
12394    hello.wasm
$ md5sum *.wasm
74118fea55ff6490d8fdd9abb201c3b1   hello_license.wasm
74118fea55ff6490d8fdd9abb201c3b1   hello.wasm
```

As you can see from the above, the compiler discards comments (in this case the license text) because the output of the two compilation commands is identical. This means that the wasm binary does not contain any of the license information, and the user who is only provided the binary cannot extract or access the license text; therefore, the user has to obtain the licensing information in another way.

There are a few ways this can be done.

Browser-based delivery of license information

If the user can interact with the WebAssembly code that is loaded on a page, then a few possible options for satisfying the license condition could include the following:

- Displaying the license information on the webpage that the user is viewing
- Displaying the license information in the readable HTML source code for the webpage (so it can be seen when a user views the source)
- Adding the license information as comments in the JavaScript code that is sent with the WebAssembly binary file
- Storing the license information in a separate file or URL, which is then linked to the web page source code or the JavaScript files
- Including an industry-standard license identifier, such as SPDX® ISO/IEC 5962:2021 the user can follow to identify the license text (note that it may additionally be necessary to include the copyright notice as well)

Textual representation code delivery

Another possibility is to generate and deliver the textual representation of the binary code and add the relevant information at the top of this file in comments (lines starting with two semicolons),

for example (example truncated for clarity):

```
;;  MIT License
;;
;; Copyright (c) <year> <copyright holders>
;;
;; Permission is hereby granted, free of charge, to
any person
[...]
```

or, using SPDX license IDs[17]:

```
;;  Copyright (c) <year> <copyright holders>
;;  SPDX-License-Identifier: MIT
[...]
```

You should note that the textual format is typically not the default mode to distribute WebAssembly programs. Using the textual format instead of the binary format might come with a performance hit, as you must first convert the textual format back to the binary format or interpret it, which is something developers will likely not want.

The above methods require that the license texts and other information be extracted first from the original source code. Of course, there is also the alternative of providing the complete and corresponding source code containing all the legal texts next to the WebAssembly binary. This removes the need to extract license texts, and for some licenses, this is possibly a better way to provide the information than first extracting the license texts and copyright notices and offering those separately. For example, for programs licensed under the GNU General Public License version 2.0 that are distributed by offering access to copy over the network, it is typically sufficient to provide "equivalent access" to the sources under GPL-2.0 (section 3) to fulfill the primary license obligations.

To recapitulate, there are the following five ways:

1. Display the license information on an (interactive) website.
2. Include license information in the webpage source code or JavaScript files, or add a separate file that is linked in the webpage source code or JavaScript files.
3. Use the textual representation, and add the corresponding license information.
4. Provide the complete and corresponding source code, and make it available next to the binary code.
5. Any of the above, but use an industry-standard license identifier referencing the specific license text.

Possible steps for compliance

Let's explore a few solutions for license compliance for distributing WebAssembly binaries.

In the first scenario, the chosen solution for compliance is that any notices and other legal texts (such as, for example, a written offer) are stored in a separate file that is stored next to the WebAssembly binary file and made available as described below.

In the second scenario, the corresponding source code including all the license texts is provided. These two scenarios are not mutually exclusive: You can provide the source code archive and separate files with all the license information and (if applicable) a written offer.

What they both have in common is that first, you need to review license obligations to see whether anything should be provided, and if so, what should be provided (only the license text or possibly more information, such as copyright notices, author notices, complete and corresponding source code, or a written offer).

Scenario 1: Separate file with license notices and other legal information

In the first scenario, provide a separate file with all the license notices and other legal information. For this, follow these steps:

1. Extract license notices, copyright statements, and author statements from the source code (either manually or using tools such as FOSSology[18], ScanCode,[19] or others).
2. Create a file or a collection of files (for example, an SBOM file in SPDX format[20] or a separate text file) with the necessary information (license texts, plus a written offer if necessary), and store them next to the WebAssembly binary.
3. Make sure that users can find the file with legal information:
    a. Include a link to the file in the HTML or the JavaScript source code.
    b. Display the link to the file with license information on the web page.

In our "Hello World" case, this means extracting the license information from "hello.c" and providing the license information as an SBOM file or in a separate file, making it available via the web server and linking it on the website or pointing to it in the HTML code.

Scenario 2: Provide complete and corresponding source code next to the binary file

In the second scenario, provide the complete and corresponding source code next to the binary file. One benefit is that the extraction of license notices, copyright statements, and author statements from the source code is not necessary, as they are already in the source code. For the rest, it isn't that much different:

1. Create an archive with the necessary source code (possibly including the build scripts, depending on the license of the source code), and store it next to the WebAssembly binary.
2. Make sure that users can find the file with the source code:
    a. Include a link to the source code archive in the HTML or the JavaScript source code.
    b. Display the link to the source code archive on the web page.

For our "Hello World" program, this means uploading the source code file to the web server and linking it on the website or pointing to it in the HTML code.

Dynamic linking and derivative works

For several licenses (particularly copyleft licenses), recurring topics in any compliance activity are understanding what constitutes derivative works and dynamic linking, so it is worth investigating whether this can also be done in the context of WebAssembly, and if so, how it can be done. Of course, whether something is a derivative of another program cannot be said without looking into the actual code itself, and no blanket statement can be given.

WebAssembly programs are not necessarily standalone programs: WebAssembly's design works in such a way that WebAssembly programs can be combined with other software, primarily JavaScript.

JavaScript wrappers

WebAssembly works with JavaScript. WebAssembly speeds up operations that are performance sensitive and for which JavaScript is not fast enough. JavaScript can manipulate the data on a webpage, passing the difficult work on to the WebAssembly code. This mechanism is not unique to JavaScript and WebAssembly. In other programming languages, it is also possible to call code written in other languages. For example, Python programs can be extended with modules written in C (this depends on the Python interpreter that is used, and it might not work or work well with every Python interpreter), Java has a mechanism called "Java Native Interface" (JNI), and so on.

The Mozilla website describes the WebAssembly JavaScript wrapping as follows[21]:

> "What's more, you don't even have to know how to create WebAssembly code to take advantage of it. WebAssembly modules can be imported into a web (or Node.js) app, exposing WebAssembly functions for use via JavaScript. JavaScript frameworks could make use of WebAssembly to confer massive performance advantages and new features while still making functionality easily available to web developers."

JavaScript code can call WebAssembly functions using a wrapper mechanism[22,23]. The wrapper mechanism first converts the data that needs to be passed from JavaScript to WebAssembly from data types used in JavaScript into data types supported by WebAssembly. It then calls the WebAssembly function in the virtual machine using the converted parameters, gets the results, converts the results back into JavaScript data types, and makes these available to the JavaScript functions, which can then further process the results.

These wrappers can be created in two different standard ways. The first way is to define a table with function references[24], fetching and loading the WebAssembly binary code and assigning the functions in the WebAssembly code to slots in the table. The second way is to first fetch WebAssembly code, load it, and then access functions of the WebAssembly module[25] that are exported to the outside world by the virtual machine. The WebAssembly functions are exported from WebAssembly to JavaScript and are called "exports."

The WebAssembly code can also access JavaScript functions. These are imported into WebAssembly from JavaScript and are called "imports."

Imported and exported functions can create a coupling between the WebAssembly and the JavaScript code. How tight this coupling is depends on the situation. There could be a minimal coupling (for example, a generic WebAssembly module that exports functions), but it is also possible to have a more intimate coupling between

the WebAssembly code and the JavaScript code. To determine whether there is a coupling to the degree that it would constitute a derivative work or a "work based on" another program, it is necessary to understand how code interacts with other code, which methods are imported and exported, and how these are used.

Dynamic linking

Code in WebAssembly can be dynamically linked. There doesn't seem to be a standard mechanism for this supported across compilers, but rather, support will be dependent on each compiler. In the context of the Emscripten compiler,[26] there is support (with some limitations) for shared modules that enable dynamic linking. The same approach may not work with other Wa compilers. Emscripten supports dynamic linking at load time (loading with the main module) or at runtime (calling side modules after running the program). Standardization in this area of WebAssembly appears to be in an earlier stage of development and will require further research to fully understand the implications.

Accessing native code from WebAssembly

Proposals for accessing specific functionality in or outside the virtual machine have been made, including the WebAssembly System Interface (WASI)[27]. WASI's aim is to provide a standardized and platform-neutral API to access functionality that is implemented or exposed by the virtual machine. These proposals are currently still in development and possibly not implemented by every virtual machine. The WASI interface gives a WebAssembly application access to host functions. For example, you could have an application write and save a file to the local host system[28].

Decompilation of WebAssembly

There are situations where the only available code is the binary code or its textual representation, and there is no source code where source code is needed (for example, for an audit, the source code is typically easier to audit than the WebAssembly binary code or its textual representation). One way to get something like source code is by decompiling the WebAssembly binary code. There are tools that can do this—for example, the `wasm-decompile` tool from the wabt package[29]. You can invoke this as follows:

```
$ wasm-decompile hello.wasm
```

The output is something that resembles a source code file, although it does not necessarily resemble the original source code at all. The original `hello.c` file is six lines, while the decompiled code is more than 1,800 lines.

This is not surprising. The WebAssembly virtual machine is on purpose fairly simple with a small set of instructions. The result is that when writing complex programs in a higher-level language, the code has to be translated to the simpler instructions of the virtual machine, which can lead to many WebAssembly instructions being generated because WebAssembly doesn't have the same expression power. Going back from the low-level instructions to a higher-level language is a lot more work, and it might also be impossible to recreate what was originally written.

WebAssembly programs can be written in various languages, such as C, C++, Rust, and others, and these are all compiled into the same WebAssembly instructions (and could also be combined into a single program). Unless extra information is kept about what language the program was originally written in, it is hard to recreate something that resembles the original program.

It could very well be that in the future, smarter or more advanced decompilers will be developed. Until then, it is wise to not rely on the decompiled output of WebAssembly programs as reflecting the original source code's structure or specific content.

Outbound licensing

So far, the focus has been on inbound licensing: what to do with code from third parties containing open source code that is reused and/or redistributed. How to license your own contributions is also something that you should consider. Depending on the license that you choose, there are also actions that you can take to make it easier for downstream recipients of your code who might want to redistribute your code themselves as well.

Pick a license

The first step is to pick a license. There are some things that you probably want to take into consideration:

1. How are the other components that you are using licensed? It makes no sense to pick a license for your code that is not compatible with the licenses of other components. A license compatibility check is recommended.

2. What obligations are there when licensing the code under a certain license?

The license you pick has consequences for downstream recipients of your code if they want to redistribute the code. It is good to be aware of how others in the same ecosystem are licensing their code. After all, this is the code that your code will most likely be combined with. In many ecosystems, there tends to be a preference for particular licenses.

Choose a standard license header

It is strongly recommended to use a standard license header instead of inventing your own. License scanners and legal professionals recognize standard license headers. Writing new versions unnecessarily complicates things. The SPDX license website[30] lists templates for common standard headers for many licenses that you can use.

Make license information available in an easy-to-access format

If you pick a license for your own code that comes with license obligations, such as disclosure of license texts or the complete and corresponding source code, it really helps your downstream recipients if you also provide a software bill of materials (SBOM), or at the minimum, provide the necessary information to create SBOM files. To create SBOM files, look at SPDX[31]. Other lightweight mechanisms for communicating a project's license information are ABOUT[32] or REUSE[33] (the latter of which is based on SPDX metadata fields).

Conclusion

WebAssembly is a relatively new technology, but this doesn't mean it can escape from the same rules for distributing software as other technologies; license terms will still apply. It is without question that open source software will be used, and this means that open source license compliance will be required when distributing WebAssembly software. Fortunately, you can leverage a lot of expertise and experience that is already available in the community.

This document explored some of the potential problem areas for WebAssembly open source compliance, such as which formats are used in the WebAssembly ecosystem, where (or where not) to put license texts and other license disclosure documents, interactions with other code (Javascript wrappers, dynamic linking) and outbound licensing. Now it is up to the WebAssembly ecosystem to come up with license compliance best practices that best serve that ecosystem.

About the author

Armijn Hemel, MSc, is a leading open source license compliance engineer hailing from The Netherlands. In the past 15 years, he has helped hundreds of companies with open source compliance, ranging from fighting copyright trolls in court to auditing software before going to market. He is also actively writing open source tools to help with open source supply chain management.

Acknowledgments

Special thanks to Luis Villa (Tidelift), Steve Winslow (Boston Technology Law), and Mike Dolan (The Linux Foundation) for providing comments and improvements to the drafts. Thanks also to the members of the legal community who support compliant technological and open source innovation in all its forms, including broadening their understanding of WebAssembly to ensure a thriving open source ecosystem across organizations and jurisdictions around the world.

Disclaimer

This report is provided "as is." The Linux Foundation and its authors, contributors, and sponsors expressly disclaim any warranties (express, implied, or otherwise), including implied warranties of merchantability, noninfringement, fitness for a particular purpose, or title related to this report. In no event will the Linux Foundation and its authors, contributors, and sponsors be liable to any other party for lost profits or any form of indirect, special, incidental, or consequential damages of any character from any causes of action of any kind with respect to this report, whether based on breach of contract, tort (including negligence), or otherwise, and whether or not they have been advised of the possibility of such damage. Sponsorship of the creation of this report does not constitute an endorsement of its findings by any of its sponsors.

Endnotes

1 https://webassembly.org/docs/use-cases/

2 "minified JavaScript" is a form of JavaScript with most markup (line feeds and so on) replaced to make the code that needs to be sent from the server to the web browsers smaller

3 https://github.com/bytecodealliance/wasm-micro-runtime

4 https://github.com/wasmerio/wasmer

5 https://webassembly.org/

6 https://developer.mozilla.org/en-US/docs/WebAssembly

7 https://emscripten.org/

8 https://pythondev.readthedocs.io/wasm.html

9 https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format

10 https://github.com/WebAssembly/design/blob/main/Rationale.md

11 https://developer.mozilla.org/en-US/docs/WebAssembly/C_to_wasm

12 A headless client is a program that doesn't graphically display results to the user.

13 https://webassembly.github.io/spec/web-api/index.html

14 https://github.com/WebAssembly/wabt

15 https://www.openchainproject.org/

16 https://spdx.org/licenses/MIT.html

17 https://www.linuxfoundation.org/blog/blog/solving-license-compliance-at-the-source-adding-spdx-license-ids

18 https://www.fossology.org/

19 https://github.com/nexB/scancode-toolkit/

20 https://spdx.dev/

21 https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts

22 https://developer.mozilla.org/en-US/docs/WebAssembly/Exported_functions

23 https://webassembly.org/getting-started/js-api/

24 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Table

25 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Instance/exports

26 https://emscripten.org/docs/compiling/Dynamic-Linking.html

27 https://github.com/WebAssembly/WASI

28 https://wasmbyexample.dev/examples/wasi-hello-world/wasi-hello-world.assemblyscript.en-us.html

29 https://github.com/WebAssembly/wabt/blob/main/docs/decompiler.md

30 https://spdx.org/licenses/

31 https://spdx.dev/

32 https://github.com/nexB/aboutcode-toolkit/blob/develop/docs/source/specification.rst

33 https://reuse.software/

THE LINUX FOUNDATION | Research

Founded in 2021, Linux Foundation Research explores the growing scale of open source collaboration, providing insight into emerging technology trends, best practices, and the global impact of open source projects. Through leveraging project databases and networks, and a commitment to best practices in quantitative and qualitative methodologies, Linux Foundation Research is creating the go-to library for open source insights for the benefit of organizations the world over.

twitter.com/linuxfoundation

facebook.com/TheLinuxFoundation

linkedin.com/company/the-linux-foundation

youtube.com/user/TheLinuxFoundation

github.com/LF-Engineering

December 2022

To reference the work, please cite as follows: Armijn Hemel, "WebAssembly (WASM) for legal professionals: Exploring current parameters in license compliance," The Linux Foundation, December, 2022.