Wassail: a WebAssembly Static Analysis Library (Extended Presentation Abstract)

Quentin Stiévenart
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
quentin.stievenart@vub.be

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.be

Keywords WebAssembly, static analysis

1 Introduction

WebAssembly is a recent web standard [4] designed to provide a portable execution environment. As of the time of writing, more than 40 languages support WebAssembly as a compilation target[1]. Initially targeted at running inside Web browsers, WebAssembly can nowadays be run in a multitude of runtimes targeting various usages, ranging from cloud applications[2] to IoT devices [1].

Tooling for WebAssembly is gaining traction. The WebAssembly website lists a number of mature tools for compiler writers[3]. The research community has produced tools to perform dynamic analysis of WebAssembly [2] and to fuzz implementations of WebAssembly [3]. In terms of static analysis, there exists a simple prototype plugin for IDA Pro to load WebAssembly modules[4], as well as a code size profiler that is able to construct call graphs[5]. However, both of these tools are not in active development.

In this presentation, we will present our work on Wassail[6], a static analysis library for WebAssembly. Our goal is to facilitate the development of various static analyses for WebAssembly, by providing a set of useful building blocks from which analyses can be constructed. We aim to support both *lightweight* static analyses such as code querying, as well as *heavyweight* static analyses such as dataflow analyses. Wassail is still in an early development phase and we are interested in feedback from the community to steer its future development.

2 Design Overview

Figure 1 depicts the current design of Wassail. Boxes indicate the main library modules made available by Wassail, while text fields represent the various datatypes that represent intermediate or final analysis results.

Given a WebAssembly file, `Wasm_module` loads the file and constructs a value of type `Wasm_module.t` that represents the static description of the loaded module. To ensure compatibility with the WebAssembly standard, all of the parsing is handled by the reference implementation provided alongside the WebAssembly standard[7], which is written in OCaml just like Wassail. Once a `Wasm_module.t` has been constructed, multiple analyses phases can be run. We describe these phases and illustrate them with real-world usage examples.

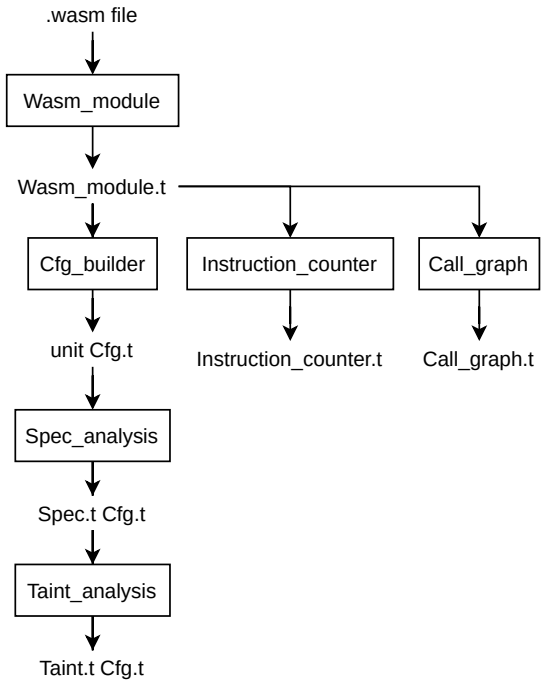

Figure 1. Design of Wassail

2.1 Counting Instructions

As a first example phase, the `Instructions_counter` module implements a lightweight analysis that solely depends on the initial description of a WebAssembly module. It counts

[1] https://github.com/appcypher/awesome-wasm-langs
[2] https://wasmer.io/
[3] https://webassembly.org/getting-started/advanced-tools/
[4] https://github.com/fireeye/idawasm
[5] https://rustwasm.github.io/twiggy/
[6] https://github.com/acieroid/wassail
[7] https://github.com/WebAssembly/spec

ProWeb21, Online, United Kingdom
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

how often each instruction appears in the WebAssembly module, and can be used for simple code inspection. This analysis can process large binaries fairly quickly: early experiments show that binaries of several megabytes can be analysed in a matter of seconds.

2.2 Call Graphs

From the WebAssembly module, it is possible to derive an approximate call graph by inspecting `call` and `call_indirect` instructions. This is what the `Call_graph` analysis phase does. Call graphs can then be exported in a visual representation, where nodes denote functions with their index, and edges denote possible calls, as depicted on the left of Figure 2.

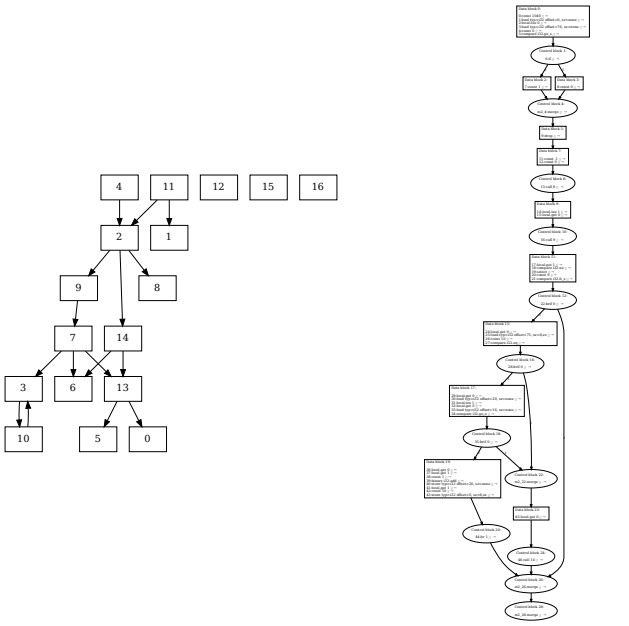

Figure 2. Example call graph (left) and control-flow graph (right) extracted by Wassail.

2.3 Control-Flow Graphs

Most heavyweight analyses require control-flow graphs to be computed beforehand. This is handled by the `Cfg_builder` module, which returns a value of type `unit Cfg.t`. Control-flow graphs are annotated with extra information, initially of type `unit` to represent the absence of annotation, which is filled by later analysis phases. The right part of Figure 2 depicts an example call graph extracted by Wassail.

2.4 Dataflow Analyses

Wassail supports expressing dataflow analyses on the CFGs computed by the previous phases. These analyses will produce a CFG annotated with the new information that has been computed. In practice, while an intra-procedural dataflow analysis can work at the level of a single CFG, inter-procedural analyses may benefit from being run on all CFGs in a bottom-up manner according to the call-graph. This is supported by Wassail as well.

Stack Specification Analysis A first dataflow analysis infers the specification of the WebAssembly stack; annotating each instruction with the shape of the stack before and after it has executed, and assigning unique names to each of the stack locations. To illustrate this, the comments among the example instructions below correspond to the output of this phase:

```
;; push 0 on the stack
i32.const 32 ;; annot: [const_0]
;; push first local on the stack
local.get 0 ;; annot: [local_0, const_0]
;; add the two top values of the stack
i32.add ;; annot: [add_0]
```

Taint Analysis Given the specification of the stack computed by the stack specification analysis, the taint analysis phase assigns to each unique name a set of *taints* denoting the information it may contain. A prior publication [5] describes the taint analysis in detail.

3 Conclusion

This presentation will cover the current design of Wassail and aims at fostering discussion on the necessary tooling infrastructure to support the needs of WebAssembly users and researchers. We will greatly appreciate feedback from the audience on how to increase the usefulness of Wassail for the community.

Acknowledgments

This work is supported by the *"Cybersecurity Initiative Flanders"*.

References

[1] Adam Hall and Umakishore Ramachandran. 2019. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI 2019, Montreal, QC, Canada, April 15-18, 2019*. 225–236.

[2] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. 1045–1058.

[3] Árpád Perényi and Jan Midtgaard. 2020. Stack-Driven Program Generation of WebAssembly. In *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings (Lecture Notes in Computer Science)*, Bruno C. d. S. Oliveira (Ed.), Vol. 12470. Springer, 209–230. DOI: http://dx.doi.org/10.1007/978-3-030-64437-6_11

[4] Andreas Rossberg. 2019. WebAssembly Core Specification. (2019). https://www.w3.org/TR/wasm-core-1/

[5] Quentin Stiévenart and Coen De Roover. 2020. Compositional Information Flow Analysis for WebAssembly Programs. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 13–24. DOI: http://dx.doi.org/10.1109/SCAM51674.2020.00007