Verifying Memory Safety Properties in Programs using WebAssembly Emscripten APIs
CS639A - Fall 2020

Baldip Singh Bijlani, Yugesh Ajit Kothari

Abstract

As web platforms continue to mature, web applications are becoming much more sophisticated and demanding. Applications like 3D visualization, rendering softwares, audio video compression decompression, are extremely computation intensive. Javascript as the only built-in language of the web is insufficient by design to meet these requirements in terms of efficiency and safety.

WebAssembly[1] (Wasm) is an upcoming low-level platform-independent byte-code language that can run on all major browsers today. This allows developers to compile C/C++ code to Wasm, and run them at near native speeds in browsers. However, this compilation preserves traditional memory safety vulnerabilities like use-after-free, null-dereference, etc.

Moreover, in the current design of running Wasm in browsers, pieces of C/C++ compiled Wasm code are glued together using Javascript and HTML and therefore there is a lot of interaction of these binaries with "external" code. This makes verification of such code, challenging beyond simply identifying classic vulnerabilities or proving safety properties as there is a high degree of false alarms. In this project, we therefore make an attempt to apply Angelic Verification[2] to such C/C++ code and demonstrate its effectiveness in verifying memory safety properties while suppressing false positives.

1. Introduction

WebAssembly is a safe, portable, low-level bytecode language and stack-based virtual machine designed for the increasingly sophisticated domain of Web applications. It offers compact representation, efficient validation and compilation, and low to no-overhead execution. It is being developed by a W3C community group. WebAssembly allows developers to take their native C/C++ code to the browser, where it can be run with near-native performance by the end user. As of this writing, all major browsers support executing WebAssembly (.wasm binaries), and it is already implemented in over 80% of all browsers on the web[3].

WebAssembly has a linear memory model in which load and store operations are performed to an untyped array of bytes. This memory model is a key feature which makes it possible for compilers like Clang to easily and efficiently compile C/C++ code to target Wasm (-march=wasm). Unfortunately, this is also the reason why memory safety vulnerabilities, like use-after-free(s) and buffer overflow(s) remain a problem when C/C++ programs are compiled to Wasm [4, 5].

Browsers currently run Wasm code in a sandbox, isolating the impact of potential security vulnerabilities in Wasm code from the rest of the browser. But sandboxing

Wasm code execution does not keep the Wasm code safe from itself. This is because isolation cannot prevent an attacker from exploiting memory-safety bugs to compromise the Wasm code and any of the data it handles - for example cryptographic keys. Therefore, as adoption of WebAssembly increases, it becomes interesting to prove safety properties at different levels of hierarchy - verifying both native C/C++ code intended to be compiled to wasm and the wasm byte-code itself. It also becomes important to guarantee memory safety properties for such code to prevent security vulnerabilities.

Cutting-edge processor features like tagged memory, pointer authentication, etc make it increasingly possible to detect, mitigate, and prevent such vulnerabilities with low overhead. Unfortunately, Wasm JITs and compilers cannot exploit these features as critical high-level information like the size of an array, is lost when lowering to Wasm.

The alternative approach to avoid any runtime overheads is to prove these properties statically. Popular tools and verifiers usually perform static analysis on an intermediate language like LLVM or Boogie[6]. Boogie is an intermediate verification language (IVL), intended as a layer on which to build program verifiers for other languages. Therefore, one possible path to proving safety properties would be to compile our C/C++ source to Boogie and proving these properties on Boogie. We can perform this verification step statically by compiling our code to Boogie, instrumenting our code with memory-safety assertions and then using a whole-program analyzer like Corral[7] to find assertion violations.

An open source compiler toolchain called Emscripten[8] has comprehensive and portable support for C/C++ standard libraries, WebGL APIs, and can be used to write C/C++ code for the web. Emscripten compiles C/C++ code to spit out wasm binaries coupled with wrapper .js scripts. Due to this, and how these scripts are eventually folded into web applications, C/C++ programs written using Emscripten APIs can be viewed as open programs that interact with the browser through external APIs. This is a problem because such static analysis can generate a lot of false positives due to a lack of modelling of these APIs that help interact with the environment. High rate of false alarms is an issue as these need to be weeded out by a developer, which is often time consuming. To overcome this issue we turn to Angelic Verification (AV). AV is a technique for verification of open programs, where we constrain a verifier to report warnings only when no acceptable environment specification exists to prove the assertion. This reduces the number of alarms and is especially helpful in suppressing false alarms.

There is very little prior work that looks at verifying WebAssembly, nor are there any good benchmarks that capture how WebAssembly is currently being used in production. To address these issues, in this project, we present a proof-of-concept demonstration of the effectiveness of Angelic Verification to verify memory safety properties while suppressing false positives. Overall we make the following contributions:

1. We demonstrate the effectiveness of Angelic Verification to verify memory safety properties in C/C++ code written as open programs using Emscripten sdk to be compiled to Wasm. To our knowledge, there is no prior work that does this.
2. We prepare a (small) labelled benchmark of curated programs which elicit the problem of using conventional verification techniques to verify such programs.

Organization. Next, in section 2, we give a brief overview of memory safety properties and relevance of Angelic Verification in the context of programs using Emscripten's

APIs. In section 3, we talk briefly about the benchmarks we collect to test the effectiveness of Angelic Verification. In section 4, we talk about our experimental setup. We discuss our evaluation in section 5 and end with our impressions of this idea in section 6.

2. Memory Safety

In this section, we discuss at a high level what memory safety bugs mean semantically and the role of Angelic Verification in this context. In C/C++ memory-safety bugs result from how different compilers interpret undefined behavior in a program. As an example, suppose a program attempts to write beyond the end of an array. One perfectly valid interpretation of this could be to crash the program. However, since array bounds checking can cause performance overheads, compilers often implement a more dangerous interpretation: just write to whatever other object is in that memory location, and simply continue running. This interpretation is dangerous because it ends up violating a programmer's assumptions about separation between different data objects[9]. When maliciously crafted inputs are given to such interpretations, they create memory-safety vulnerabilities, as the programmer has inadvertently given an input control over unintended parts of program data and control flow.

In practice, to avoid behaviour that violates this separation, we need to ensure three memory-safety properties[10]:

1. **pointer integrity** - disallows pointers from being created from non-pointer values (e.g., casting a non-pointer primitive data type to a pointer)
2. **temporal safety** - blocks exploitation of use-after-free
3. **spatial safety** - blocks out-of-bounds reads and writes

Together, these properties ensure that every pointer dereference in a C program returns data from a valid object. The overhead of implementing these checks in pure software is often high; even optimized JIT-based approaches can incur more than 2× performance overhead for enforcing full memory safety[11]. Thus, static analysis techniques are of importance especially in performance critical applications where unnecessary overhead cannot be tolerated. However, static analysis is incomplete, has limited effectiveness without pervasive source-code annotations and suffers from high false positive rate which programmers then need to weed out manually.

In our evaluation, we find that Angelic Verification is able to infer a "weakest liberal precondition" in most cases thereby reducing false positives. This technique is therefore highly useful in domains where our verification task is poised to have a high degree of false alarms, as is the case of C/C++ programs that use Emscripten APIs because of a lack of modelling of these APIs.

3. Benchmarks

In order to explore the use of Angelic Verification in the domain of WebAssembly , we collected a set of benchmark programs written in C/C++ that use emsdk to interact with browser APIs and other related functionaliy like XHR and IndexDB. These

programs capture how small real world programs would look like and how they would use external APIs to interact with the environment.

Due to a lack of prior work in this context and limited open source projects, finding such benchmarks **was extremely difficult and time intensive**[1]. We have adopted most of our programs from the Emscripten test-suite and modified some of them slightly to suit our purpose. Given below is an incomplete list of programs, along with their API families and nature of alarm, that appear in our benchmarks. We have tried to ensure uniqueness for an <API-family, nature> pair to collect as wide an array of behaviour as we were able to. For example, some alarms are generated because the implementation of an API was in JavaScript which corral could not incorporate, while some are because of a lack of modelling - like if an object is allocated memory and populated behind an API call.
In section 5, we note however, that a major limitation of this work is the size of our benchmarks.

File	API Family	Nature of bug (alarm)
sdl2_swsurface.c	SDL	False invalid dereference
test_em_js.cpp	EM_JS(for writing javascript code in CPP files)	Missed memory leak (true bug)
html5_webgl.c	html5_webgl	False invaild free
test_html5_mouse.c	html5	False invalid deref in callback function
idb_delete.cpp	fetch	False invalid dereference due to lack of API model
websocket.c	websocket	False invalid deref
test_gamepad.c	html5	False invalid deref in callback function
custom_em_js_test.c	EM_JS	False invalid deref

4. Experimental Setup

The experimental setup for this project consists of the entire smack toolchain and the emscripten software development kit (emsdk). To make our experiments reproducible, we provide a Dockerfile with the rest of our code and benchmarks for easy setup. All of our scripts, benchmarks, stubs and experiments are publicly available[2].

[1] We spent close to 80 human hours scouring through programs to identify ones that would fit our purpose.
[2] https://github.com/yugeshk/CS639

4.1 SMACK Toolchain

SMACK is both a modular software verification toolchain and a self-contained software verifier. It can be used to verify the assertions in its input programs. SMACK compiles C/C++ code to Boogie via LLVM. Targeting Boogie exploits a canonical platform which simplifies the implementation of algorithms for verification, model checking, and abstract interpretation. As a toolchain, Smack comes bundled with Boogie and Corral verifiers. Corral is a whole-program analysis tool for boogie programs. Corral uses goal-directed symbolic search techniques to find assertion violations in input programs. It leverages the powerful SMT solver Z3. The latest version of Smack requires LLVM 10.1. As mentioned, smack toolchain comes coupled with Corral and Boogie. We use Z3 as our default SMT solver, although it is possible to use yices2 or CVC4 as well.

4.2 Emscripten and EMSDK

Emscripten is a compiler that compiles C/C++ to WebAssembly using LLVM and Binaryen. The emscripten sdk comes bundled with support for multiple libraries like webgl, opengl, etc. It also redefines a lot of the header files in the standard C library. Emscripten then exposes APIs that are either wrappers over these libraries or depend upon them. Developers can write their C/C++ code using these APIs which are then compiled to WebAssembly.

4.3 AngelicVerifier

Independent of the default smack toolchain, for the Angelic Verification experiments we also set up AngelicVerifierNull.exe which was distributed as a Corral AddOn till Corral v1.0.15. For the purposes of this experiment we set up AngelicVerifierNull.exe distributed with Corral v1.0.12. The AV setup is also part of the docker build.

5. Evaluation

5.1 Experiments

To evaluate the effectiveness of AngelicVerification in suppressing false alarms, we run experiments in three steps which are as follows:

- **Run Corral Without Models or AV** : The first step is to simply run the benchmark programs simply with Corral to see how many bugs (assertion violations) it finds. To ensure correctness, and since our test-bench is small we also manually inspected the violations, to find if they are true bugs or false bugs.
- **Run Corral With Models** : There are broadly two kinds of models. Stubs and Harnesses.
    - Stubs are simplified implementations of emscripten APIs. We need these as these APIs are for providing C++ support for capabilities that are specific to JavaScript or the browser environment, and often their source code is written in JavaScript, or there is inline JavaScript in the C/C++ code. Since there is no frontend for JavaScript to boogie translation, we

need to manually define these stubs. Below we give an example of a stub we wrote for inline JavaScript code.

- EM_JS(line 6) is used to define functions in JavaScript and call them in C. On line 6 a function called get_unicode_str is defined which takes no parameters and returns a const char pointer. It allocates some memory on the Heap which is a JavaScript array when it is running in the browser, puts a string in that memory and returns it. To write a stub for the same we translate it to C code as shown on line 16, so that this can be translated to boogie and be verified by corral. In the absence of this stub the output of get_unicode_str on line 26 will be unconstrained and a false null-free bug is identified, which in the presence of this stub does not arise.

```
1  #include <emscripten.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  EM_JS(const char*, get_unicode_str, (), {
7    var jsString = 'Hello with some exotic Unicode characters:
       T ss  on yksi lumiukko:    , ole hyv .';
8    // 'jsString.length' would return the length of the string as UTF
       -16
9    // units, but Emscripten C strings operate as UTF-8.
10   var lengthBytes = lengthBytesUTF8(jsString)+1;
11   var stringOnWasmHeap = _malloc(lengthBytes);
12   stringToUTF8(jsString, stringOnWasmHeap, lengthBytes);
13   return stringOnWasmHeap;
14 });
15
16 const char* get_unicode_str(){
17 char jsString[100] = "Hello with some exotic Unicode characters: T
       ss  on yksi lumiukko:  ^x^c, ole hyv  .";
18 int lengthyBytes = strlen(jsString);
19 char *stringOnHeap = (char*)malloc(sizeof(char)*lengthyBytes);
20 return stringOnHeap;
21
22 }
23
24
25 int main() {
26   const char* str = get_unicode_str();
27   printf("UTF8 string says: %s\n", str);
28   // Each call to _malloc() must be paired with free(), or heap
       memory will leak!
29   free(str);
30   return 0;
31 }
```

Figure 1 : Source code for test_em_js.cpp : example program from our benchmarks.

- Harnesses are required as there are callback functions defined which are not called anywhere in the main program but are for handling events which happen in the browser environment, eg.mouse click. If you verify these functions, assuming that their inputs are arbitrary, you will find

that there are numerous false positives, as the context in which these handlers are called ensure that their input parameters obey certain properties,eg. a pointer is non null, an integer parameter is the length of the buffer pointed to by a pointer parameter, etc, but these are unknown to the verifier.

- **Run AV** : While running AV, no models as mentioned above need to be provided. AV treats the output of functions without implementations, as angelic, i.e. it will try to find a specification on the said output such that it is able to block a failing infeasible(would not be allowed if the environment was modelled accurately) program execution while ensuring that possible true bugs are not suppressed, with the help of certain heuristics.

After running each program in the benchmark set with all three methods we report the number of false positives, false negatives and true positives. We also remark on the cases for which AV works well, and for the cases for which it does not.

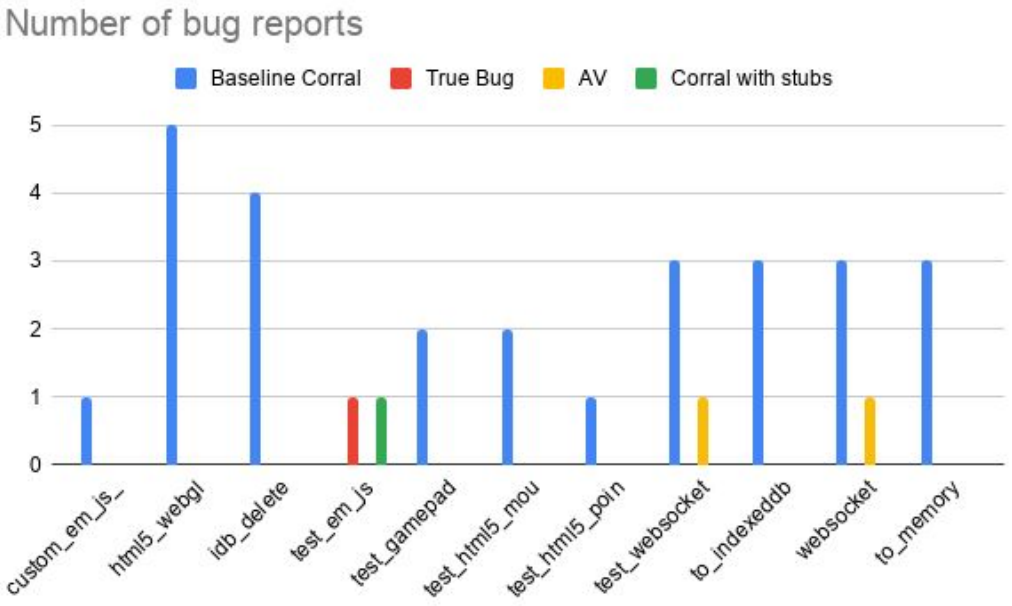

Figure 2 : Shows number of alarms across different experiment steps. All colours apart from red are false alarms.

5.2 Results

Figure 2 depicts a comparison of the number of reported alarms with baseline corral (corral without models), corrals with stubs and AV. Benchmarks are indicated on the X-axis, number of alarms raised are indicated on the Y-axis, and the number of true bugs in the benchmarks is indicated in red. From the figure we observe two things :

1. **Using AV we are able to suppress almost all false alarms.** The false alarms which were identified as null-dereferences by vallila corral, were suppressed by

AV. Although a few false alarms which were identified as buffer overflows were not suppressed by AV.

2. **AV is unable to capture a true positive memory-leak**, in the file test_em_js.cpp. This is because memory is allocated in inlined JavaScript code, and is not freed. Though using a model(stub for the inlined JavaScript code), written in C, corral identifies a memory leak. Though AV cannot come up with a very complex assumption that memory is allocated inside a procedure for which the source code is not given.

Limitations. The programs that we used as benchmarks, were tests in the emscripten repository and not real real world programs. They did not have very complex logic and were short in length. Hence they did not have many true bugs. Due to this we were not able to ascertain that AV is able to suppress false positives while preserving true positives or, does AV very aggressively suppress all errors. We had manually seeded one true bug and a couple of false bugs we knew would be reported by corral.

6. Conclusion

Even though static analysis techniques are incomplete and laborious on the part of the developer, they are important in keeping low runtime overhead for safety property checks. They can be used to find most of the security and reliability defects during development, and checks can be inserted in the code automatically to detect some of the remaining errors at runtime. In this project we make an initial attempt in exploring the use and effectiveness of Angelic Verification to C/C++ programs that use the EMSDK APIs for compilation to WebAssembly. As motivated in previous sections, we observe that with the way Emscripten works, such programs are prone to having high degree of false alarms due to a lack of modelling of the external APIs, some of which may not even necessarily be implemented in C/C++. In our evaluation, we find that Corral's Angelic Verifier is indeed able to suppress a lot of the false alarms (by proving more assertions) which are otherwise generated because of a lack of modelling of the APIs. **We therefore conclude from our preliminary experiments that Angelic Verification is indeed suitable for helping the developer in his verification endeavour.**

We note however, that programs in our benchmark were relatively simple, with shallow bugs. We therefore acknowledge that more testing on a wider array of real-world programs is needed in this application to ascertain AV's performance, possibly even exploring custom AV grammars developed specifically for this use case. We leave these as future work.

7. Future Work

We identify the following aspects under future work to further elicit the usefulness of Angelic Verification in this application domain. Foremost, there is a need for more thorough testing of AV on a wider array of real world programs with more nested bugs, and true bugs, to ascertain that AV is able to balance suppressing false bugs while still preserving true bugs. Further, through such rigorous testing, a possible outcome could be that the current AV grammars(s) are insufficient, in which case another possible direction is exploring grammar specific to this use case.

8. Acknowledgements

This work was done in 6 weeks as part of our project for CS639 Program Analysis, Verification and Testing, taught by Prof. Subhajit Roy in the Fall semester of 2020 at IIT Kanpur. We thank Prof. Subhajit Roy for the wonderful course he has taught and the opportunity it gave us to explore Angelic Verification and a promising new application for the same. We thank Prof. Subhajit Roy and Prantik Chatterjee for their inputs in concretizing this idea and giving it shape. We thank Prantik Chatterjee for all the technical help he has provided promptly. Lastly we thank Dr. Akash Lal at Microsoft Research for giving good direction in understanding the entire Smack toolchain and Corral's AngelicVerifier AddOn.

References

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the Web up to speed with WebAssembly. PLDI, 2017.
[2] Das A., Lahiri S.K., Lal A., Li Y. (2015) Angelic Verification: Precise Verification Modulo Unknowns. In: Kroening D., Păsăreanu C. (eds) Computer Aided Verification. CAV 2015. Lecture Notes in Computer Science, vol 9206. Springer, Cham. https://doi.org/10.1007/978-3-319-21690-4_19
[3] A. Deveria. Can I use WebAssembly?, 2019. https://caniuse.com/#feat=wasm.
[4] B. McFadden, T. Lukasiewicz, J. Dileo, and J. Engler. WebAssembly: A new world of native exploits on the browser. In Blackhat briefings 2018, 2018.
[5] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell. Exploring C semantics and pointer provenance. POPL, 2019.
[6] K. Rustan M. Leino. 2012. Program proving using intermediate verification languages (IVLs) like boogie and why3. Ada Lett. 32, 3 (December 2012), 25–26. DOI:https://doi.org/10.1145/2402709.2402689
[7] Lal A., Qadeer S., Lahiri S.K. (2012) A Solver for Reachability Modulo Theories. In: Madhusudan P., Seshia S.A. (eds) Computer Aided Verification. CAV 2012. Lecture Notes in Computer Science, vol 7358. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-31424-7_32
[8] Zakai, Alon. (2011). Emscripten: an LLVM-to-JavaScript compiler. 301-312.
[9] A. A. de Amorim, C. Hritcu, and B. C. Pierce. The meaning of memory safety. arXiv:1705.07354, 2017.
[10] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. 2019. Position Paper: Progressive Memory Safety for WebAssembly. In Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '19). Association for Computing Machinery, New York, NY, USA, Article 4, 1–8. DOI:https://doi.org/10.1145/3337167.3337171
[11] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. POPL '02. ACM, 2002.
[12] P. Hickey. Announcing Lucet: Fastly's native WebAssembly compiler and runtime, Mar 2019.
[13] K. Varda. WebAssembly on Cloudflare workers, Dec 2018.