École Polytechnique Fédérale de Lausanne

Pitfalls and characterization of C libraries isolation through WebAssembly

by Vincenzo Pellegrini

Master Project Report

Msc. ETH Andrés Sánchez Marín
Project Supervisor

Prof. Dr. sc. ETH Mathias Payer
Professor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 5, 2024

Abstract

This project explores the issues faced when trying to automate library isolation of C libraries using WebAssembly; using type analysis on library headers, as well as runtime analysis on simple drivers, it is shown that C features such as recursive structs, unions and function pointers are widely used in libraries, and that fields of library structs are accessed from the driver.
In a second part, a possible approach to face this problems is shown, by directly accessing the shared memory of the WebAssembly module in the driver, through automatic llvm level instrumentation of the driver using static analysis.

Contents

Chapter 1

Introduction

Isolation of programs from third-party libraries written in C remains a critically modern problem in software engineering. While there's a plethora of tools available for this task, they often require significant developer engagement and understanding of both the library and the isolation tool. This translates into a substantial upfront engineering cost, especially when using tools like RLBox[1], which necessitate detailed, manual intervention for effective isolation.

The complexity of current solutions stems partly from the nature of C itself—a language that offers extensive control over memory management and system resources but at the cost of increased security risks when integrating third-party libraries. These libraries, while powerful, can potentially introduce vulnerabilities or system instabilities, making isolation not just a performance issue but also a critical security concern.

This project explores a different approach to address these challenges. By leveraging the capabilities of WebAssembly (Wasm), we aim to automate the isolation process of C libraries, reducing the burden on developers. WebAssembly offers a promising avenue due to its design for secure, sandboxed execution and its growing support in various environments. Our solution investigates the feasibility of automating library isolation using Wasm, focusing on reducing the manual effort required while maintaining high performance and security standards.

In the following sections, we delve into the specifics of this approach, examining both its theoretical underpinnings and practical implementations. We will explore how WebAssembly can be harnessed to create a more streamlined, automated process for library isolation, potentially transforming how developers interact with third-party C libraries in a secure and efficient manner.

Chapter 2

Background

In this chapter, the problem at hand is presented, one of the possible solutions and its tradeoffs, and finally introduce some useful tools that are used in the project.

2.1 Limitations of current approaches

Current approaches to software fault isolation often require careful commitment from developers of end application to deliver a working result, take for example RLBox. While this solutions are used in the wild successfully (RLBox[1], NaCl[3]), their upfront cost may lead to lower adoption rates.

Gobi[2] is an early prototype that enables library isolation through Wasm. While this work mentions solutions to the memory sharing problem, by modifying the WASM backend of the compiler, the solution presented does not handle modifications to the driver automatically, that therefore has to determine which allocations should be made on the shared memory.

2.2 Problems with mirroring approach

Previous works such as PtrSplit[0] require memory serialization and deserialization at each library function call. Using an in-process sandboxing technique such as WebAssembly, it's possible to share memory objects between the driver and the isolated library.

Sharing memory in such a way brings many challenges that are explored in this project, however it is a priority when using Wasm. Here we list some of the problems with serialization in a C program:

- **Recursive objects**: Pointers that contain pointers to other objects require maintaining large

mappings between source and destination addresses, in order to correctly map the complex relationships between objects; in fact determining at compile time whether the pointer graph of say a linked list contains any loop would be a very complex problem.

- **Unions**: Unions in C are not tagged, therefore the type of the object at compile time is not known. This becomes a problem when a union contains multiple data structures that have a different alignment of pointers, or pointers and normal values at the same offset, as, without any further information, the correct deserialization of such objects is undetermined.

2.3 SVF

SVF[0] is a static analysis tool for C/Cpp, capable of full pointer analysis of large programs, as well as query directed analysis, among other features. It is as a large Cpp library, which takes LLVM bytecode on which it does its analysis.

In this project it was used to implement Andersen Analysis of full programs, giving access both to points-to and reverse-points-to information for each pointer in the program.

2.4 WebAssembly

Wasm[0] is a portable binary code format for executable programs. It was released in 2017 to enable native-like performance for web applications and browser plugins; isolation and high performance were a big part of the motivation behind it.

The Wasm virtual machine is a stack machine with a pointer width of 32 or 64 bits, depending on the architecture variant. In this project we only considered the 32 bit variant, called wasi-32, because of limitation in the WASI-SDK (the compilation toolchain used, more on this later).

Libraries are usually compiled to modules, which export functions to be called from the driver as well as importing functions from the driver; it is possible for modules to import functions from other modules, but this is not currently supported by most compilation toolchains.

Linear memory

Linear memory is a continuous buffer of bytes that can be read and modified by both Wasm and the driver. Normally, it contains all of the memory regions used by the wasm module (stack and heap), as well as to pass objects between the driver and the module. Pointers in the wasm module are, in fact, indexes of this array.

The linear memory can grow and shrink during execution, but this is subject to configuration, as usually this is undesirable, therefore a larger memory region is **reserved** by the driver before allocating the linear memory to its initial size.

WASM C ABI

An **Application Binary Interface** is an interface between two binary programs. Because of clear differences between native binaries and Wasm modules, a different ABI is adopted by WASM.

By examining the ABI[0], it is clear that the most significant difference is pointer size and alignment. This however transitively effects sizes and alignments of structs and arrays which contain pointer, as well as the alignment and offset of their fields.

In addition, structs passed as values and returned as values are transformed into additional pointer arguments in the function signature, requiring the object to be passed or returned to be allocated on linear memory.

2.4.1 WASI-SDK

WASI sdk[0] is a toolchain used to compile C/Cpp code to Wasm, also providing an implementation of libc (called libc). It is mentioned here as it **does not** support the *wasm-64* as well as the*multi-memory* proposals, leading to certain design choices and limiting what extensions can be created to support isolation of multiple modules independently.

2.4.2 WasmTime

WasmTime[0], a WebAssembly runtime written in Rust, facilitates the integration of Wasm modules into C programs. It provides a simplified API for calling exported functions from these modules.

One notable feature of WasmTime is its support for Ahead-of-Time (AOT) compilation. This reduces startup overhead significantly, given that the library's Wasm file is available at compile time.

Moreover, WasmTime enables the specification of the reserved memory size for a module's linear memory at instantiation. This feature is crucial for ensuring that the linear memory bounds remain constant post-instantiation, a key assumption used in the rewriter part of this project.

Chapter 3

Design

3.1 Type analysis on libraries

As mentioned in the introduction, certain C language features are problematic when trying to achieve library isolation through Wasm, in particular when adopting a memory copying approach.

A tool was created to detect the presence of problematic types in C libraries automatically. In particular, this tool detects the existence of

- Recursive structs
- Unions
- Function pointers as parameters

It first uses a parser to extract all struct, union and function definitions from the header file, to then apply fixed point analysis algorithm to detect dependency loops in struct definitions. A fixed point analysis is also used to determine which functions take function pointers as parameters, either directly or inderictly, through structs.

The use of union types is also reported.

3.2 Automatic library isolation with WebAssembly and memory sharing

To achieve automatic sandboxing of the library with WASM, the library is first compiled to Wasm using Wasi-SDK; it is then embedded in a C module (that exposes its interface) using Wasmtime with Ahead of Time compilation enabled, for better startup performance. Finally, the driver is instrumented to move part of the allocations to shared memory, as well as modifying memory accesses to shared memory to respect the different memory layout.

The prototype developed is split in two different CLIs, one that generates a **C** wrapper, based on the header file, that exposes the functions defined in the library's header, while using **WasmTime** to execute them under isolation. The other cli takes the llvm bitcode of the driver and outputs rewritten bitcode.

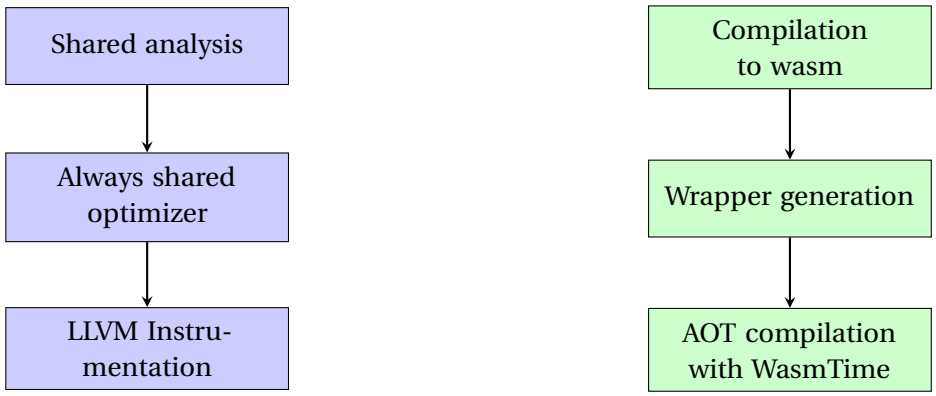

Figure 3.1: Phases of automatic library isolation

The rewriter first determines which allocations and which llvm values are on shared memory, to then run an optimization algorithm that determines which values are guaranteed to be in shared memory, and finally instrument the driver's bitcode in order to modify allocations and memory accesses accordingly.

3.2.1 Principles of the memory sharing approach

In a memory sharing approach to library isolation, all memory that can be accessed (read and written) by the library in a correct execution must be allocated on the linear memory, which is accessible both by the driver and the library; furthermore, the layout of structs and arrays allocated on the shared memory is the one used by the library (in this case, WASM32).

As a consequence of the premise of all legally accessible memory being accessible by the library, and as a direct consequence of the memory layout of objects allocated on the shared memory, all pointers in shared memory must point to shared memory.

Listing 3.1: Example struct in C

```
struct node {
    struct node *next;
    int value;
};
```

Let's consider, for example, the struct *node* from listing 3.1. While in **x64** C ABI an object of type node will have a size of 128 bits and an alignment of 64 bits, **WASM32** the same object will have a size of 64 bits and an alignment of 32 bits. The alignment of the fields will also differ accordingly.

It's important to note that only the driver has to be modified, while the library code does not need any modifications, apart from being wrapped in a WebAssembly runtime and some binding code to issue function calls.

3.2.2 Compile-time classification of pointers

Using pointer analysis, all pointers that may live in share memory are identified, by applying a set of rules. In a second phase, this set is further divided to identified pointers that are certain to be on shared memory, leading to a more lightweight instrumentation at those specific memory accesses.

Chapter 4

Implementation

4.1 Type analysis

The type analysis tool was implemented in Rust, and leverages the **LibClang api of Clang** to parse the library's header.

After parsing the header, only structs and union that are used as parameters and return types of functions in the header files are used; functions from the standard library are excluded.

References to structs have to be treated with special care because of the particularity of the Clang's AST, which only mentions the struct's name when it encounters a typedef, instead of refering to the previously defined struct.

The algorithm that finds recursive structs uses fixed point. A similar algorithm also determines which functions take functions as parameters.

4.2 Automatic memory sharing isolation

The library wrapper codegen was not completed, therefore the rewriting part of the tool couldn't be tested.

4.2.1 Pointer coloring

To determine which values may be on shared memory, and which allocations need to be done on the shared memory, a fixed point algorithm is used, first determining an initial set of values and allocations, based on the actual arguments and return values of library functions callsites, and then by propagating on the llvm module based on a limited set of rules.

Initialization of shared nodes

As a first step, for each of the call sites of library functions, the points-to set of the arguments is inspected, and the values which are allocations are marked as shared allocations. All nodes in the reverse points-to set of this allocations are then marked as shared.

Furthermore, the return values of shared functions are also marked as shared.

Propagation of shared nodes

The following rules are used to create an over-estimate (a superset) of the nodes that could be on shared memory:

- **Loads** from shared memory are shared (as pointers in shared memory **must** point to shared memory
- The value argument of **stores** for which the pointer argument may be shared **must** be shared. In this case all allocations of the value argument must happen on shared memory, and their reverse points-to set is also shared.
- **GEPs** of shared nodes are also shared, as they point to fields or elements of arrays in shared memory.
- **Copies**, **PHIs**, **Selects** and **bitcasts** of shared nodes are also shared, as it's evident from their definition.

There's also a less obvious rule that is **necessary** in order to achieve completeness of this algorithm: it's important to handle the case in which a shared value from shared memory is stored in

another value that is not; because of limitations of SVF (we cannot look at the reverse points to set of an allocation if the allocation is not done in the driver, for example if the node was marked as shared because it was the result of a load from another shared node) we may need to handle this case manually.

The adopted solution is to find the allocations of the pointer parameter of the store, then mark all loads from nodes pointed to by the allocation of those allocations as shared.

$$\frac{f \in \text{libFs} \quad a \in ptsTo(p)}{a = malloc(...) \in \text{sharedAllocs}} \qquad \frac{v \in revPtsTo(a) \quad a \in \text{sharedAllocs}}{v \in \text{shared}} \qquad \frac{*p = v \quad p \in \text{shared}}{v \in \text{shared}}$$

$$\frac{p = \quad (p.f) \quad p \in \text{shared}}{p \in \text{shared}} \qquad \frac{y = x \quad x \in \text{shared}}{y \in \text{shared}} \qquad \frac{y = bitcast_T(x) \quad x \in \text{shared}}{y \in \text{shared}}$$

$$\frac{z = phi(x,y) \quad x \in \text{shared} \vee y \in \text{shared}}{z \in \text{shared}} \qquad \frac{*p = v \quad v \in \text{shared} \quad p \notin \text{shared} \quad a \in ptsTo(p)}{a \in containsSharedAlloc}$$

$$\frac{v \in revPtsTo(a) \quad a \in \text{containsSharedAlloc}}{v \in \text{containsShared}} \qquad \frac{v = *p \quad p \in \text{containsSharedAlloc}}{v \in \text{shared}}$$

Figure 4.1: Shared propagation rules

Always shared node optimization

To improve the performance of the rewritten driver, it's useful to determine which pointers are always on shared memory, as opposed to ones that could be on normal memory.

To do this, it's important to node that all allocations done in the driver are nodes in SVF. By examining the points to set of each of the marked node, the tool determines which ones point to nodes that are not allocated on the shared memory, and subsequently all the other nodes that were marked before but can directly or indirectly point to one of those. This is achieved by collecting the points-to set of each of the nodes marked as shared, de facto constructing a subgraph of the points-to graph, limited to the node that might be shared. By using a dfs on the reverse of this graph, all nodes that point to other nodes outside of the subgraph, directly or indirectly, are individuated.

The pointers in the set extracted by the previous algorithm are not guaranteed to point to shared memory at runtime, while the rest are.

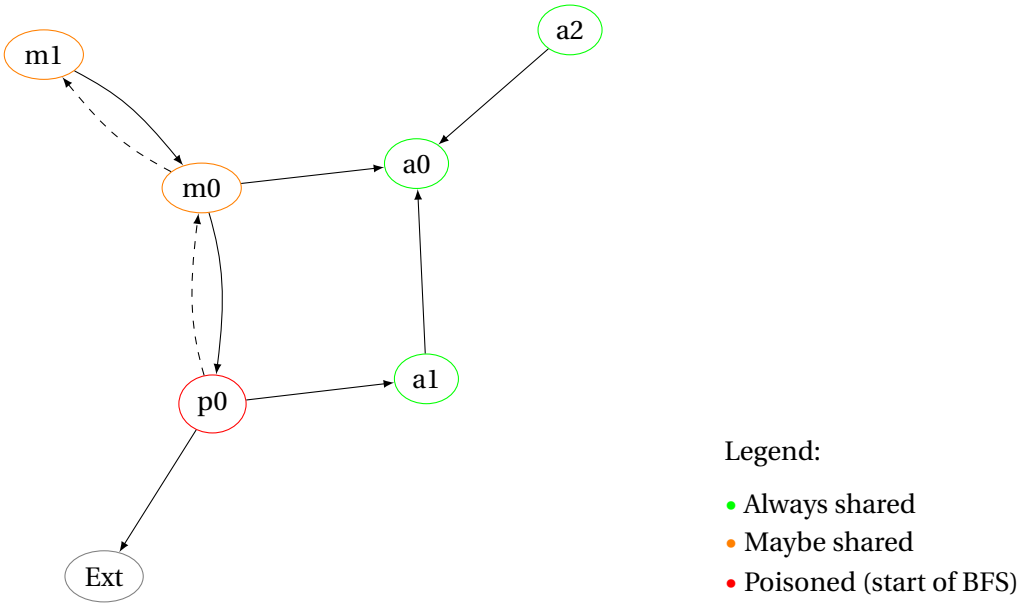

Figure 4.2: Visualization of always shared optimization algorithm

4.2.2 Driver instrumentation

Allocations

Allocations that were previously determined to happen on shared memory are substituted with a call to a function in the wrapper.

Other instructions

For the instructions to come the code generated is different wheter the pointer is certain to live on shared memory or not. If the point might not point to shared memory, a check is inserted to only execute the modified operation if the pointer is part of shared memory, by comparing its location with the bounds of the linear memory. If, instead, the pointer is guaranteed to live on shared memory, this check is not necessary.

Loads

Loads from shared memory are modified if the result type is pointer: since wasm pointers are indexes on the linear memory, the pointer is bitcast to an integer pointer, the pointed index is then loaded and summed with the linear memory's base address to determine the actual location of the pointed object.

Stores

Similarly, stores of pointers to shared memory should save the index in linear memory, so a subtraction is performed before storing the difference, truncated, as an i32.

GEPs

GEPs have to be instrumented to deal with the different memory layout of WASM32 memory. If the GEP's pointer operand is an array of pointers, the element type is changed to i32. If, instead, the GEP returns a field of a struct, an aptly modified struct is created, with fields modified to use i32 instead of pointers and, if other structs or constant length arrays of pointers are contained as value, their modified version. The GEP is then done using the modified struct as element type, and the result is bitcast to the original return type.

4.2.3 Library wrapper codegen

Library wrapper codegen was not completed by the end of the project. That said it's a simpler problem to solve, as the wrapper can be generated just from the header.

The following functions are created inside the wrapper:

- An inizialization function, that instantiate the WebAssembly module, extracts all its exported functions, creates the linear memory and takes note of its bound in **global variables**;
- A function each for
  - Allocation
  - Freeing objects that are certain to be on shared memory (saves one bounds check)
  - Freeing objects that may be on shared memory
- A function to cleanup the wasm engine.
- For each function of the untrusted library, an implementation that calls the corresponding extracted function of the module, eventually allocating memory and freeing it in case of structs passed by value (see Background on **WASI ABI**)

Chapter 5

Evaluation

5.1 Type analysis on

The type analysis tool was run on some C libraries to prove the use of language features problematic for isolation (recursive structs, unions, functions pointers) in the interfaces of C libraries.

Table 5.1: Type Analysis of C Libraries

Library Name	Recursive Structs	Function Pointers	Unions
libjpeg-turbo	Yes	Yes	No
libpng	Yes	Yes	No
libvpx	Yes	No	No
theora	No	No	No
vorbis	Yes	Yes	No
zlib	No	Yes	No

5.2 Automatic isolation of C libraries

The implementation of the wrapper generator tool was not completed, therefore no performance metrics could be collected. However it was tested on some simple C programs during the development to confirm that the expected values were detected as shared and that the instrumentation was done as expected.

This is not however a validation of the validity of the transformation, as that would require both a full implementation of the wrapper codegen and a more extensive test set with automated verification that certain values are classified as expected.

Chapter 6

Related Work

RLBox

RLBox[1] is a toolkit for sandboxing third party C libraries, currently used in production; among other sandboxing techniques it can use Wasm. Unfortunately this tool requires an effort from the developer, who has to write a wrapper for the library manually, although it does permit adding bounds checks and other validations of the output data of third party libraries, making the result more secure in principle.

TRust

TRust[0] is a framework for automatic isolation of unsafe Rust code. While its range of application is different, it has similarities as it leverages SVF for pointer analysis, as well as isolating C libraries. That said our solution does not rely on special hardware features as TRust does, furthermore accesses from the driver to memory that is in the shared memory region need heavy modifications.

Gobi

Gobi[2] is an early work on library isolation through Wasm. While the idea of allocating objects in a shared memory region was explored, and it even proposed a solution to the alignment issue we faced, the paper does not provide any information about *automatic* changes to the driver, but only mentions exposing an API to translate Wasm pointers to ones that can be used in the driver. One must then assume that no automation was devised on that front, possibly rendering it less usable than other solutions like **RLBox**, which also provide utilities to access pointers (by relying on language features of Cpp). *Note though that this work was published earlier than RLBox.*

SFI based solutions

SFI requires the instrumentation of library code, with the addition of bound checking at every untrusted access; WebAssembly however promises better performance compared to SFI, while not relying on any hardware specific features or modifications to the OS (which some of the alternatives require). The solution proposed here instead moves the cost to memory accesses in the driver, hoping that just a limited subset of pointers of a program are deemed to be shared with the library and therefore suffer the cost of the instrumentation.

Chapter 7

Future work

This project brings some interesting techniques and ideas to the table in regards to C library isolation with WebAssembly.

However, further work is required to bring the automated isolation to its fruition, in particular the **wrapper codegen** has to completed.

In addition the whole solution requires **functional testing** as well as **benchmarking** to compare it with preexisting solutions and native execution.

Finally, the **analysis currently requires a full pointer analysis** of the entire driver. This is due to time constraints on the development, however this solution would not scale to large codebases; further, the reverse points-to set of some memory locations (in particular ones that are allocated by the isolated library) cannot be accessed by SVF because of limitations in its design. A more sophisticated pointer reachability analysis, probably based on sparse value flow graph, would be required.

Chapter 8

Conclusion

In this project, we discussed the challenges posed by the C language when trying to implement memory isolation of third-party libraries using WebAssembly. Through type analysis of library headers, we were able to detect the use of types problematic for serializations in C libraries, showing their use in many commonly use C libraries. In a second phase, we proposed our solution to some of the problems shown, by sharing a memory region between the wasm module and the driver, and modifying the driver at LLVM bytecode level to deal with the coexistence of multiple memory architectures, as well as pointers referring to different base addresses, in the same C program. SVF was leveraged to identify uses of pointers to shared memory and their allocation, in a partial implementation.

Bibliography

[0] Inyoung Bang, Martin Kayondo, HyunGon Moon, and Yunheung Paek. "TRust: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6947–6964. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/bang.

[0] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. "Bringing the web up to speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 185–200. ISBN: 9781450349888. DOI: 10.1145/3062341.3062363. URL: https://doi.org/10.1145/3062341.3062363.

[0] Shen Liu, Gang Tan, and Trent Jaeger. "PtrSplit: Supporting General Pointers in Automatic Program Partitioning". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2359–2371. ISBN: 9781450349468. DOI: 10.1145/3133956.3134066. URL: https://doi.org/10.1145/3133956.3134066.

[1] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. "Retrofitting Fine Grain Isolation in the Firefox Renderer". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 699–716. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/narayan.

[2] Shravan Narayan, Tal Garfinkel, Sorin Lerner, Hovav Shacham, and Deian Stefan. "Gobi: WebAssembly as a Practical Path to Library Sandboxing". In: *ArXiv* abs/1912.02285 (2019). URL: https://api.semanticscholar.org/CorpusID:208637454.

[0] Yulei Sui and Jingling Xue. "SVF: interprocedural static value-flow analysis in LLVM". In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. Barcelona, Spain: Association for Computing Machinery, 2016, pp. 265–266. ISBN: 9781450342414. DOI: 10.1145/2892208.2892235. URL: https://doi.org/10.1145/2892208.2892235.

[0] *WASI SDK repository*. Accessed 2024. URL: https://github.com/WebAssembly/wasi-sdk.

[0] *Wasmtime*. Accessed 2024. URL: https://wasmtime.dev/.

[0] WebAssembly Community Group. *Wasm C ABI*. Accessed 2024. URL: https://github.com/WebAssembly/tool-conventions/blob/main/BasicCABI.md.

[3] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code". In: *2009 30th IEEE Symposium on Security and Privacy*. 2009, pp. 79–93. DOI: 10.1109/SP.2009.25.