BREWasm: A General Static Binary Rewriting Framework for WebAssembly

Shangtong Cao[1⋆], Ningyu He[2⋆], Yao Guo[2], and Haoyu Wang[3]✉

[1] Beijing University of Posts and Telecommunications, China
[2] Key Lab on HCST (MOE), Peking University, China
[3] Huazhong University of Science and Technology, China
haoyuwang@hust.edu.cn

Abstract. Binary rewriting is a widely adopted technique in software analysis. WebAssembly (Wasm), as an emerging bytecode format, has attracted great attention from our community. Unfortunately, there is no general-purpose binary rewriting framework for Wasm, and existing effort on Wasm binary modification is error-prone and tedious. In this paper, we present BREWasm, the first general purpose static binary rewriting framework for Wasm, which has addressed inherent challenges of Wasm rewriting including high complicated binary structure, strict static syntax verification, and coupling among sections. We perform extensive evaluation on diverse Wasm applications to show the efficiency, correctness and effectiveness of BREWasm. We further show the promising direction of implementing a diverse set of binary rewriting tasks based on BREWasm in an effortless and user-friendly manner.

Keywords: WebAssembly, Binary Rewriting

1 Introduction

WebAssembly (Wasm) [58], endorsed by Internet giants like Google and Mozilla, is an assembly-like stack-based low-level language, aiming to execute at native speed. Portability of Wasm is achieved by the ability of being a compiling target for mainstream high-level programming languages, e.g., C/C++ [61], Go [48], and Rust [31]. Lots of resource-consumed and -sensitive software have been compiled to Wasm binaries and embedded in browsers [2], like 3D graphic engines and scientific operations Beyond the browser, Wasm is moving towards a much wider spectrum of domains, e.g., IoT [29], serverless computing [18], edge computing [36], and blockchain and Web 3.0 [15].

The rising of Wasm has attracted massive attention from our research community. As an emerging instruction format, our fellow researchers have invested huge effort into Wasm binary analysis, e.g., static analysis for vulnerability detection [7], dynamic analysis based on program instrumentation [25], Wasm binary transformation [10], and binary optimization [9]. More or less, most existing studies rely on Wasm binary rewriting to achieve their goals. For example,

⋆ The first two authors contribute equally. Haoyu Wang is the corresponding author.

Wasabi [25] is a dynamic analysis framework against Wasm, which obtain the runtime information of target binaries via instrumenting. However, due to the case-specific demands of existing work, researchers need to implement a specific set of rewriting rules from scratch, or even manually modify Wasm binaries, which are error-prone. We, therefore, argue that a general purpose rewriting framework is necessary to facilitate the research on Wasm binaries.

Binary rewriting is a general technique to modify existing executable programs, which is a well-studied direction for native binaries [33,16,13,59,30]. Unfortunately, there currently is no general-purpose binary rewriting framework for Wasm. Implementing such a rewriting framework is challenging. First, Wasm is unreadable and complicated in syntax. The low-level nature of such an assembly-like language makes it extremely hard to reason about its original intention. Though Wasm formally explains the functionalities of its 11 valid sections, the syntax of them is highly structured and varies. As a user-friendly general rewriting framework, it should handle both the unreadability and the syntactic complexity in a concise way, which is a natural contradiction. Second, modifying a functionality in Wasm may require updating several sections accordingly. For example, if a user intends to insert a new function, he has to update several sections simultaneously. Manually updating all sections is fallible. Third, Wasm enforces a strict verification before executing, while any violations against the Wasm syntax during rewriting Wasm binaries will invalidate them. An invalid Wasm binary cannot be loaded and executed at all. Therefore, the binary rewriting on syntactic level cannot be conducted in an arbitrary way.

This Work. In this paper, we implement BREWasm, a general-purpose binary rewriting framework for Wasm, consisting of: *Wasm parser*, *section rewriter*, *semantics rewriter*, and *Wasm encoder*. Specifically, the Wasm parser and encoder are implemented based on our abstraction of Wasm binaries. Based on these abstracted objects, the section rewriter is able to conduct fine-grained rewriting, e.g., inserting/deleting a new object. The semantics rewriter further combines them and offers another set of high-level APIs, where each of them possesses rich semantics, like inserting a function. Thus, Wasm binaries can be arbitrarily modified without considering the underlying complexity of syntax.

Based on benchmarks consisting of representative Wasm binaries, the evaluation results show the efficiency, correctness, effectiveness, and real-world usability of BREWasm. Specially, it is practical to achieve various kinds of complicated Wasm binary rewriting tasks by combining the APIs provided by BREWasm, including binary instrumentation, binary hardening, and mutation-based fuzzing. Comparing with the cumbersome implementation of these specific tasks, the work built on BREWasm is effortless and user-friendly.

Our contribution can be summarized as follows:

- To the best of our knowledge, we have implemented the first general purpose Wasm binary rewriting framework, named BREWasm, which offers more than 31 semantic APIs that are summarized from real-world usage scenarios. It offers new insights for the design of binary rewriting tools.

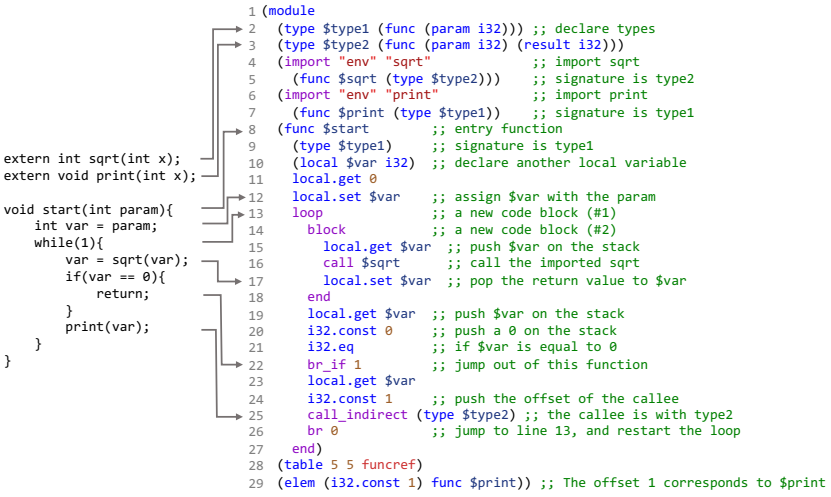

Fig. 1. A code snippet of WebAssembly in WAT text format with its source code.

- We perform extensive evaluation on diverse Wasm applications to show the efficiency, correctness and effectiveness of BREWASM.
- We show that it is useful, effortless, scalable, and user-friendly to implement a diverse set of binary rewriting tasks based on BREWASM.

To boost further research on Wasm binary rewriting, we release the implementation of BREWASM as well as the corresponding documentation at link.

2 Background

2.1 WebAssembly Binary

WebAssembly (Wasm) is an emerging cross-platform low-level language that can be compiled from various programming languages, e.g., C/C++ [61], Rust [31], and Go [48]. Wasm is designed to be effective and compact. It can achieve nearly native code speed in performance with a sufficiently small size (100KB to 1MB for an ordinary Wasm binary [32]). In addition, some official auxiliary tools are proposed to facilitate the development of Wasm. For example, *wasm2wat* can translate a Wasm binary into WebAssembly Text Format (WAT for short), and *wasm-validate* [51] can validate the syntax of a Wasm binary.

Each Wasm binary is composed of sections with different functionalities. Specifically, in a Wasm binary, functions are implemented in the ***code section***, and their signatures are declared in the ***type section***. The ***function section*** maintains a mapping from the index of each function to the index of its corresponding type. Functions can also be imported from environment through the ***import section***, and be exported via the ***export function***. Except for their independent contexts, data stored in the ***global section***, ***data section***, and

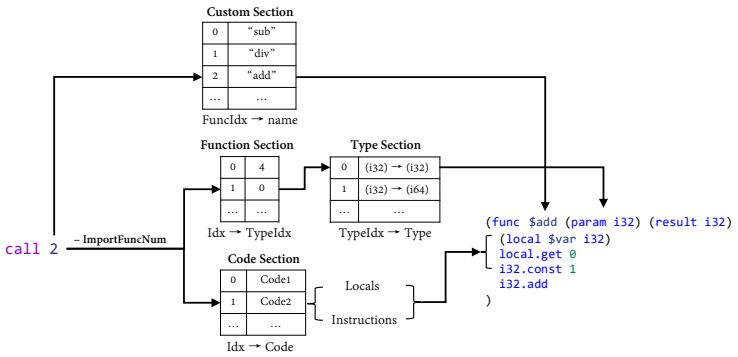

Fig. 2. Function indexing is achieved by coupling several sections.

memory section can be accessed arbitrarily. To implement function pointers, Wasm designs an *indirect call mechanism*, where all possible callees have to be declared in the ***table section*** and ***element section*** (also denoted as elem section). Additionally, debugging information or third-party extensions will be stored in the ***custom section***, which has no effect on execution.

Sections can be further divided into *vectors*, the smallest unit that declares a functionality. For example, Fig. 1 shows a code snippet of Wasm binary (shown in WAT text format) as well as its source code in C. As we can see, L2[4] is a vector belonging to type section, which declares a function signature. A vector may have many attributes, e.g., the vector at L3 consists of its index (`$type2`), parameters type (`i32`), and the return value type (`i32`). In Wasm, however, *a single semantics is often achieved by coupling several sections*. Taking indexing a function as an example, which is shown in Fig. 2. Once an internal function, indexed by 2 in this example, is invoked by a `call` instruction, its readable name can be indexed via the custom section. To obtain its signature, we need a two-layer translation through the function section and the type section. Its implementation can be accessed only via the code section. Note that, all imported functions are located in front of normal functions, thus we need to subtract the number of imported functions, which is 1 in this example, to obtain its real index when indexing in the function section and the code section.

2.2 Binary Rewriting

Binary rewriting[5] refers to the process of taking a binary as an input, rewriting various parts of it, and generating another binary that is properly formatted. The semantics of the rewritten program depends on the rewriting purpose and strategy. This technique has been widely adopted in the software analysis direction, e.g., program instrumentation [28,41,8,22], binary enhancement [37,60,23], and program transformation [10,47,5].

[4] The second line, denoted by L2. We adopt such notations in the following.
[5] In this work, the *binary rewriting* specifically refers to the static binary rewriting.

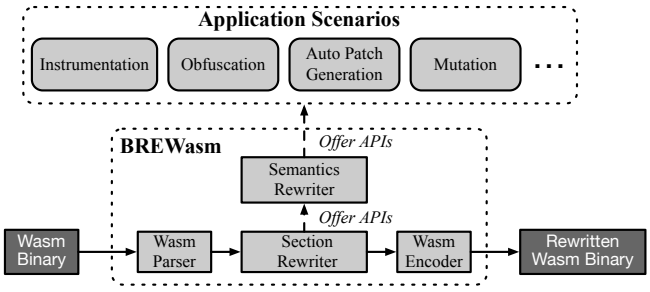

Fig. 3. The architecture and workflow of BREWasm.

Currently, some work specifically conducts rewriting against Wasm binaries. For example, Wasabi [25] is a dynamic analysis framework of Wasm. It can dynamically obtain the runtime information of the target binary via instrumenting Wasm instructions. Fuzzm [27] statically inserts stack canaries into the linear memory to identify memory bugs by conducting fuzzing. SEISMIC [52] also conducts instrumentation against Wasm binaries to determine whether the target is a malicious mining program. At last, Wasm-mutate [4] is a binary mutation tool. It integrates many different strategies for performing mutation on Wasm binaries, e.g., deliberately inserting functions or a piece of memory. All of these tools rely on binary rewriting to implement the core functionalities. However, due to their case-specific demands, developers need to implement a specific set of rewriting rules from scratch, or even manually modify Wasm binaries.

3 BREWasm

3.1 Overview

To the best of our knowledge, we have implemented the first general rewriting framework against Wasm binaries, dubbed as BREWasm, whose architecture and workflow are shown in Fig. 3. As we can see, BREWasm is composed of four components: *Wasm parser*, *section rewriter*, *semantics rewriter*, and *Wasm encoder*. Specifically, the Wasm parser takes a Wasm binary as an input, and parses it as a list of objects. Based on these objects, the section rewriter is able to conduct rewriting, e.g., inserting/deleting a new object or modifying attributes of existing objects. It packs these fine-grained rewriting functions into four basic APIs. The semantics rewriter combines these fine-grained APIs and offers another set of high-level APIs, where each of them possesses rich semantics, like inserting a function, and appending a piece of linear memory. Through these exposed APIs, users can rewrite Wasm binaries for different goals, e.g., obfuscation, instrumentation, or patch generation on vulnerabilities. Finally, these updated objects will be encoded into a valid Wasm binary through the Wasm encoder. The implementation of BREWasm is detailed in §4, and some usage scenarios will be depicted in §5.5.

3.2 Challenges

Implementing a general rewriting framework against binaries is often challenging. Because binaries are highly structured and have little semantic information to guide the rewriting process. As for Wasm binaries, as we introduced in §2.1, although Wasm supports inter-conversion between binary and text formats, it is still infeasible to directly modify its text format for rewriting purposes. This can be attributed to three points, i.e., *complicated format*, *strict static verification*, and *coupling among sections*. We detail these three challenges in the following.

C1: Format Complexity. Conducting efficient and effective static binary rewriting is strongly correlated to the complexity of the rewritten binary. From Fig. 1, we can observe that Wasm is a syntactic complicated format. Specifically, as we introduced in §2.1, there are 11 valid sections defined under the current specification [56]. Each section is composed of vectors, each of which is further composed of several attributes. As an assembly-like language, each attribute is indispensable and corresponds to a specific and unique meaning. Moreover, these sections are highly structured. For example, for a vector in type section (see L3 in Fig. 1), the attributes `param` and `result` are wrapped by a `func`, which is further wrapped by the corresponding `type`. The same situation also plays for other sections. Therefore, the syntactic complexity of Wasm makes it exceptionally challenging to implement a general rewriting framework.

Our Solution: In order to enhance readability and facilitate the following rewriting process, we implement a parser to translate the given Wasm binary into an array of *objects*, each of which is composed of several *attributes*. During the parsing process, we also omit some auxiliary strings, like the `type` and the `func` at L3 of Fig. 1. Therefore, each highly structured and nested vector will be translated into an object with several side-by-side attributes according to its semantics. Binary rewriting can be easily performed by modifying objects. Meanwhile, we also implement an encoder to conduct the opposite process, i.e., translating objects into a Wasm binary. Please see §4.1 for more details.

C2: Static Verification. Each Wasm binary will be thoroughly and strictly verified statically before executing [17,57]. Such a static verification performs on several aspects. For example, sections are composed of vectors, each of which is indexed by an index. Thus, if a user inserts/deletes a vector, he has to update vectors of the whole section to ensure the continuity of indices of vectors. Moreover, as we mentioned in **C1**, the encoder has to reassemble objects and complete the auxiliary strings that are discarded in the parsing process. Any negligence will invalidate the rewritten Wasm binary.

Our Solution: To solve this problem, we implement a fixer that will be automatically invoked after each time of invoking the APIs exposed by the section rewriter. Specifically, the fixer is mainly responsible for repairing the incontinuity for indices of the rewritten section. It can also fix some context-aware errors, like increasing the limitation field (if necessary) to hold a newly inserted memory. After the encoding process, the official syntactic checker, wasm-validate, is invoked to examine the validity. Please refer to §4.2.

C3: Sections Coupling. As we mentioned in §2.1, some functionalities should be achieved by combining multiple sections. For example, if we add an extra function in a Wasm binary, except for inserting its implementation in the code section, we have to modify the function section and the type section to declare its signature. Moreover, if the added function can be taken as the callee of a function pointer, the table section and the elem section should also be updated accordingly. Manually updating them is tedious and error-prone. Such a section coupling raises another challenge for the user to achieve his intended goal.
Our Solution: To address the sections coupling problem, we abstract the coupling between sections into a set of Wasm program semantics, i.e., *global variable*, *import and export*, *linear memory*, *function* and *custom content*. Based on these five semantics, we expose a set of APIs, e.g., `insertInternalFunc`, which takes a function's body and signature as inputs. Inside the API, we determine if the signature already exists, and insert a new one if necessary. Then, we will insert its declaration and implementation in the function section and the code section, respectively. It is worth noting that if any reference relation goes wrong due to indices mismatch, like the index of callee of a `call` is incremented by 1 due to inserting a function, these reference relations will be repaired automatically after each time of invoking APIs of the semantics rewriter. Please refer to §4.3.

4 Approach

In this section, we will introduce the technical details of components of BREWasm, and how we address the aforementioned challenges.

4.1 Wasm Parser & Wasm Encoder

Wasm has a highly structured and complicated format. Specifically, a Wasm binary is composed of *sections*, which is a vector of *elements*. Further, an element consists of several *fields* according to the section where it locates. As we mentioned in **C1**, to facilitate the following rewriting process, we hide unnecessary and verbose details and translate Wasm binaries in a semantically equivalent format. Referring to the official Wasm specification, we formally define the relationships among sections, elements, and fields as shown in Fig. 4.

As we can see, each section is composed of a list of elements with the same name, where elements are composed of several fields. Specifically, each *custom* element is composed of a set of *index-name* pairs. These pairs can be parsed as debugging information for different purposes, like keeping readable names of functions, global variables, and a piece of data. Moreover, each *type* element consists of three fields, indicating the function signature $type_{param}{}^* \rightarrow type_{result}{}^*$ is declared by the idx_{type}-th type element. The definitions of *import* elements and *export* elements are similar. An *import* element indicates the idx_{func}-th function with the type declared by idx_{type} is imported from *modulename* and named as *name*, while an *export* element refers to the idx_{func}-th function is exported as *name*, which can be invoked by the environment. Note that, an

Section, Element & Field

$$
\begin{aligned}
\mathit{Section}\ S &::= \mathit{element}^{+} \\
\mathit{element} &::= \mathit{custom} \mid \mathit{type} \mid \mathit{import} \mid \mathit{function} \mid \mathit{table} \mid \mathit{memory} \\
&\quad \mid \mathit{global} \mid \mathit{export} \mid \mathit{start} \mid \mathit{elem} \mid \mathit{code} \mid \mathit{data} \\
\mathit{custom} &::= (\mathit{idx}_{\mathit{func}|\mathit{global}|\mathit{data}}\ \mathit{name})^{*} \\
\mathit{type} &::= \mathit{idx}_{\mathit{type}}\ \mathit{type}_{\mathit{param}}{}^{*}\ \mathit{type}_{\mathit{result}}{}^{*} \\
\mathit{import} &::= \mathit{idx}_{\mathit{func}}\ \mathit{module}\ \mathit{name}\ \mathit{idx}_{\mathit{type}} \\
\mathit{function} &::= \mathit{idx}_{\mathit{func}}\ \mathit{idx}_{\mathit{type}} \\
\mathit{table} &::= \mathit{min} \mid \mathit{min}\ \mathit{max} \\
\mathit{memory} &::= \mathit{min} \mid \mathit{min}\ \mathit{max} \\
\mathit{global} &::= \mathit{idx}_{\mathit{global}}\ \mathit{type}_{\mathit{val}}\ \mathit{mut}\ \mathit{val} \\
\mathit{export} &::= \mathit{idx}_{\mathit{export}}\ \mathit{name}\ \mathit{idx}_{\mathit{func}} \\
\mathit{start} &::= \mathit{idx}_{\mathit{func}} \\
\mathit{elem} &::= \mathit{idx}_{\mathit{elem}}\ \mathit{offset}\ \mathit{idx}_{\mathit{func}}{}^{*} \\
\mathit{code} &::= \mathit{idx}_{\mathit{func}}\ \mathit{local}^{*}\ \mathit{instruction}^{*} \\
\mathit{data} &::= \mathit{idx}_{\mathit{data}}\ \mathit{offset}\ \mathit{initData} \\
\mathit{local} &::= \mathit{idx}_{\mathit{local}}\ \mathit{type}_{\mathit{val}} \\
\mathit{instruction} &::= \mathit{op}\ \mathit{operand}^{*}
\end{aligned}
$$

Types & Literals

$$
\begin{aligned}
\mathit{type}_{*} &::= \texttt{i32} \mid \texttt{i64} \mid \texttt{f32} \mid \texttt{f64} \\
\mathit{mut} &::= \texttt{0x00} \mid \texttt{0x01} \\
\mathit{module} \mid \mathit{name} \mid \mathit{initData} &::= \texttt{byte}^{*} \\
\mathit{op} \mid \mathit{operand} &::= \texttt{byte} \\
\mathit{idx}_{*} \mid \mathit{val} \mid \mathit{min} \mid \mathit{max} \mid \mathit{offset} &::= \texttt{u32}
\end{aligned}
$$

Fig. 4. Formal definition of sections, elements, and fields in Wasm.

function element only declares a function's index and its signature, where the implementations are defined in the corresponding *code* element. The *min* and *max* defined in *memory* elements jointly limit the available size of the linear memory, and each *data* element declares that the initial value (*initData*) of the idx_{data}-th linear memory starts from the designated $offset$. Similarly, *table* elements and *elem* elements share the pattern, but $idx_{func}{}^{*}$ refers to callee indices for `call_indirect` instructions. Finally, a *global* element declares its value as *val* with type of $type_{val}$, where *mut* indicates whether its value can be updated by instructions. Two extra terms *local* and *instruction* are defined that are nestedly adopted in *code* elements.

For each section, we have defined a class with its fields as attributes. Each element is an object of the corresponding class. To this end, the Wasm parser is able to translate a Wasm binary into a list of objects. For example, Listing 1.1 illustrates the parsed objects of the Wasm binary in Fig. 1.

As we can see from Listing 1.1, the Wasm parser translates each element and packs them into their corresponding objects. The field names are hidden, but we can obtain the corresponding value according to the definition in Fig. 4. For example, L8 is a *code* element, its first parameter 2 indicates that it corresponds to the implementation of the second function. According to L6, its type can be indexed by 0, i.e., returns nothing but takes an `i32` (declared at L2). The

second field of the code element is a list of *local* objects, each of which declares the type and the value of local variables. Similarly, the third field declares all its instructions, which are wrapped by `Instruction` objects. The first instruction has an *op* valued as `0x20` and an *operand* as `0`, corresponding to the `local.get 0` at L11 in Fig. 1. Through the opposite direction, the Wasm encoder can translate and reassemble these objects into a Wasm binary in a lossless way.

```
1  parsedWasm = [
2      Type(0, ["i32"], []),
3      Type(1, ["i32"], ["i32"]),
4      Import(0, "env", "sqrt", 0),
5      Import(1, "env", "print", 0),
6      Function(2, 0),
7      # omit following instructions
8      Code(2, [Local(0, "i32")], [Instruction("0x20", [0]),
           ...]),
9      Table(5, 5),
10     Elem(0, 1, [1])
11 ]
```

Listing 1.1. Parsed objects of the Wasm binary of Fig. 1.

4.2 Section Rewriter

The section rewriter plays a vital role for BREWASM. It provides four basic APIs that allow users to manipulate sections on fine-grained level, i.e., *select*, *insert*, *delete*, and *update*. Through combining these four operations, users are able to manipulate any elements or fields we mentioned in §4.1. The syntax for these operations is formally expressed as follows:

$$
\begin{aligned}
\mathit{select} &: \mathit{element}_{\mathit{template}} \rightarrow \mathit{element}^{*} \\
\mathit{insert} &: \mathit{element}^{*} \times \mathit{element}_{\mathit{new}} \rightarrow \texttt{true|false} \\
\mathit{delete} &: \mathit{element}^{*} \rightarrow \texttt{true|false} \\
\mathit{update} &: \mathit{field} \times \mathit{field}_{\mathit{new}} \rightarrow \texttt{true|false}
\end{aligned}
$$

Specifically, within the context after parsing the given Wasm binary, the *select* takes an element template ($\mathit{element}_{\mathit{template}}$) to filter out all elements conforming to the $\mathit{element}_{\mathit{template}}$. Note that the wild card, an understrike character, is allowed when designating a field. For example, `select(Type(_, _, ['i32']))` returns all *type* elements that return a single `i32` without considering their arguments. To this end, the object at L3 in Listing 1.1 instead of the one at L2 will be returned. Based on the selected results, the *insert* and *delete* can be conducted to insert a new element ($\mathit{element}_{\mathit{new}}$) after the designated one(s) and delete the given elements, respectively. Take a concrete situation as an instance. Against Listing 1.1, if a user wants to delete the first type element and insert a

new one, with arguments as `i64` and returns as `i32`, he can write:

```
delete(select(Type(0, [‘i32’], [])))
insert(select(Type(_, _, _))[-1], Type(_, [‘i64’], [‘i32’]))
```

, where the first statement deletes the object at L2 by an exact match, and the second statement inserts a new type element after the last one. Moreover, the *update* can be used to modify a field by a new value ($field_{new}$). Field values can be retrieved by a dot operator, like getting values of an attribute in an object. For example, the user intends to modify the returns as `i64` on the just inserted type element. He can invoke the following statement:

```
update(select(Type(_, [‘i64’],[‘i32’])).resultType,[‘i64’])
```

, where the field $type_{result}$ is accessed by a dot operator with an identical name.

Though we can ensure the flexibility of them, the challenge **C2** still occurs and has to be addressed. For example, the *max* in *memory* elements declares the maximum available space for the corresponding linear memory. It is possible to update the *initData* in a *data* element resulting in exceeding the limitation. Another example is that inserting a new or deleting an existing element from any sections may lead to the incontinuity of indices. Both of these situations invalidate the Wasm binary. Therefore, we implement a fixer that is automatically invoked after each rewriting requests. The fixer can determine which section should be fixed, identify the bugs resulting from rewriting requests, and fix them. Therefore, **C2** can be addressed after a flexible rewriting process.

```
1  # suppose params, results, locals, and instrs are given by
       the user
2  # insert the type element
3  funcType = select(Type(_, params, results))
4  if funcType:
5      typeIdx = funcType[0].typeIdx
6  else:
7      insert(select(Type(_, _, _))[-1], Type(_, params,
           results))
8      typeIdx = select(Type(_, _, _))[-1].typeIdx
9  # insert the function element
10 insert(select(Function(_, _))[-1], Function(_, typeIdx))
11 # insert the code element
12 insert(select(Code(_, _, _))[-1], Code(_, local, instrs))
```

Listing 1.2. Append a function through APIs offered by the section rewriter.

4.3 Semantics Rewriter

Though the section rewriter allows users to rewrite elements and even their fields on a fine-grained level without considering the indices continuity, users still have to make effort to deal with the *section coupling problem* (see **C3**). For example,

Table 1. APIs exposed by the semantics rewriter.

Semantic	Sections	Representative API(s)	Explanations
Global Variables	Global	`insertGlobalVariable` *idx* : u32 *valType* : i32|i64|f32|f64 *mut* : 0x00|0x01 *initValue* : u32	`globalItem = Global(idx, valType, mut, initValue)` `insert(select(Global(idx, _, _)), globalItem)`

Import & Export	Type
	Import
	Export
		`insertImportFunction`
		idx : u32
		moduleName : byte*
		funcName : byte*
		paramsType : (i32|i64|f32|f64)*
		resultsType : (i32|i64|f32|f64)*
			`# insert the type element`
			`functype = select(Type(_, paramsType, resultsType))`
			`if functype:`
			`  typeidx = functype.typeIdx`
			`else:`
			`  insert(select(Type(_, _, _))[-1], functype)`
			`  typeidx = len(Type(_, _, _))[-1].typeIdx`
			`# insert the import element`
			`importFunc = Import(_, moduleName, funcName, typeidx)`
			`if select(importFunc):`
			`  pass`
			`else:`
			`  insert(select(Import(idx, _, _, _)), importFunc)`

		`insertExportFunction`
		idx : u32
		funcName : byte*
		funcidx : u32
			`# insert the global element`
			`exportFunc = Export(_, funcName, funcidx)`
			`if select(exportFunc):`
			`  pass`
			`else:`
			`  insert(select(Export(idx, _, _)), exportFunc)`

Linear Memory	Memory
	Data
		`appendLinearMemory`
		pageNum : u32
			`memory = select(Memory(_, _))`
			`if memory.max != 0:`
			`  memory.max += pageNum`

		`modifyLinearMemory`
		offset : u32
		bytes : byte*
			`# modify the initData of data`
			`for data in select(Data(_, _, _))`
			`  dataEnd = data.offset + len(data.initData)`
			`  if data.offset <= offset and offset < dataEnd:`
			`  pre = data.initData[:offset - data.offset]`
			`  post = data.initData[offset - data.offset + len(bytes):]`
			`  data.initData = pre + bytes + post`
			`  return`
			`# otherwise, insert the data element`
			`data = Data(_, offset, bytes)`
			`insert(select(Data(_, _, _))[-1], data)`

Function	Type
	Function
	Code
	Start
	Table
	Element
		`insertInternalFunction`
		funcidx : u32
		paramsType : (i32|i64|f32|f64)*
		resultsType : (i32|i64|f32|f64)*
		locals : *local**
		funcBody : *instruction**
			`# insert the import element`
			`functype = Type(_, paramsType, resultsType)`
			`if select(functype):`
			`  typeidx = functype.typeIdx`
			`else:`
			`  insert(select(Type(_, _, _))[-1], functype)`
			`  typeidx = len(Type(_, _, _))[-1].typeIdx`
			`# insert the function element`
			`insert(select(Function(funcidx, _)), Function(funcidx, typeidx))`
			`# insert the code element`
			`code = Code(funcidx, locals, funcBody)`
			`insert(select(Code(funcidx, _, _)), code)`

		`insertHookFunction`
		funcidx : u32
		hookedFuncIdx : u32
		funcBody : *instruction**
		paramsType : (i32|i64|f32|f64)*
		resultsType : (i32|i64|f32|f64)*
		locals : *local**
			`# insert the hook function`
			`hookFuncidx = insertInternalFunction(funcidx,`
			`  paramsType, resultsType, localVec, funcBody)`
			`# Modify call instruction`
			`callInstr = Instruction(Call, hookedFuncidx)`
			`newCallInstr = Instruction(Call, hookFuncidx)`
			`importFuncNum = len(select(Import(_, _, _, _)))`
			`for funcidx in range(importFuncNum, len(select(Code(_, _, _)))):`
			`  if funcidx != hookFuncidx:`
			`  funcInstrs = select(Code(funcidx, _, _)).instrs`
			`  for instr in funcInstrs:`
			`  if instr == Instruction('call', hookedFuncidx)`
			`  instr.operands = hookFuncidx`
			`code = select(Code(funcidx, _, _))`
			`code.insert(code.select(Instruction(_, _)[offset]), instrs)`

		`modifyFunctionInstr`
		funcIdx : u32
		offset : u32
		instrs : *instruction**
			`code = select(Code(funcidx, _, _))`
			`# Delete old instructions and insert new ones`
			`code.delete(code.select(Instruction(offset, _, _)))`
			`code.insert(code.select(Instruction(offset, _, _)), instr)`

		`appendFunctionLocal` *funcidx* : u32 *valType* : i32|i64|f32|f64	`code = select(Code(funcidx, _, _))` `code.insert(code.select(Local(_, _))[-1], Local(_, valType))`
Custom Content	Custom	`modifyFunctionName` *funcidx* : u32 *name* : byte*	`funcName = select(Custom(funcidx=idx, _))` `update(funcName, Custom(funcidx=idx, name))`

the indices mismatch occurs when indices for type elements are changed but still referred by elements in other sections. Fixing them manually is fallible. Moreover, to achieve a little complex functionality, it may be inconvenient for users to solely adopt such fine-grained APIs offered by the section rewriter. Take appending a function as an example, which is shown in Listing 1.2.

As we can see, L3 firstly checks if the given signature has been declared. If it is, its index is kept (L5), or a new index is calculated by inserting it into the type section (L7 to L8). Then, the user has to manually link the type index to a function index by inserting a function element (L10). Finally, the implementation of the function is appended (L12). We can see that all newly inserted elements have no concrete *idx*, which is because the fixer we mentioned in §4.2 can automatically calculate these indices.

From the instance, we can conclude that rewriting or updating a functionality of a Wasm binary always needs a series of combinations of APIs exposed by the section rewriter. To ease the burden on users and improve the usability, BREWasm provides another rewriter, named *semantics rewriter*. We have conducted a comprehensive survey in real-world scenarios on applications that require binary rewriting as the prerequisite. The survey has covered lots of representative papers [27,25,52,10] and popular repos on GitHub [4,54]. Consequently, as shown in Table 1, we have abstracted 5 semantics, which cover all 11 sections and offer 31 APIs in total that can be used by these applications. Specifically, the *global semantics* allows users to arbitrarily update values that can be accessed under the global scope. Through the *import & export semantics*, users can import or export designated functions. The *memory semantics* can be used to insert another piece of linear memory with a piece of initiated data, while the *function semantics* mainly focuses on updating functions to achieve some goals. Finally, through the *custom semantics*, users can update the debug information to perform obfuscation by changing names of functions. For example, the 12-LOC listing 1.2 can be abstracted to:

```
appendInternalFunction(params, results, locals, instrs)
```

Though it is practical to enhance the usability of implementing some functionalities through calling these 31 APIs, **C3** requires another fixer that investigates and maintains reference relations between elements across sections. Therefore, after APIs have been invoked, another fixer will be automatically waked to iterate sections and fix reference relations. Take the `appendInternalFunction` as an instance, it will examine if any element in other sections has to be fixed.

We argue that the semantics rewriter is not limited by the scalability issue. These APIs are concluded from real-world scenarios, which can satisfy most needs. Moreover, it is intuitive and practical to implement case-specific semantics APIs by users themselves through combining four APIs in the section rewriter. The three challenges can be properly handled without their intervention. On the one hand, as for **C1**, the Wasm parser can translate the given Wasm binary with complicated syntax into a list of objects (see §4.1). On the other hand, the fix against **C2** and **C3** is automatically conducted.

5 Implementation & Evaluation

5.1 Implementation

We have implemented BREWasm with over 4.3K LOC of Python3 code from scratch. To avoid reinventing the wheel, some relied modules are based on open-source GitHub projects. For example, integer literals in Wasm are encoded by LEB128 algorithm [1]. To accelerate the encoding and decoding process, we utilize the highly efficient cyleb128 library [39] implemented by Cython. We have packaged BREWasm into a standard Python library, which can be easily accessed and used by developers.

5.2 Research Questions & Experimental Setup

Our evaluation is driven by the following three research questions:

- **RQ1** Is it efficient to conduct Wasm binaries rewriting through APIs exposed by BREWasm?
- **RQ2** Whether the APIs provided by BREWasm are implemented correctly and effectively?
- **RQ3** Can BREWasm be easily applied to real-world scenarios?

To answer these questions, we first selected 10 Wasm binaries from Wasm-Bench [21], a well-known micro benchmark that collects tens of thousands of Wasm binaries. The basic information for them is shown in Table 2. We believe that these 10 binaries are representative. Specifically, they are compiled from various mainstream programming languages, and cover two typical domains of applying Wasm binaries, i.e., web scripts and standalone applications. Moreover, they range in size from 3KB to 4MB, which can effectively reflect the ability of BREWasm to handle Wasm binaries with different sizes. Though we can add more Wasm binaries as candidates for the following evaluations, some APIs (e.g., `insertHookFunction`) require our manual effort to determine concrete parameters. Thus, considering such a trade-off, we argue that these 10 Wasm binaries are representative for the whole ecosystem.

All experiments were performed on a server running Ubuntu 22.04 with a 64-core AMD EPYC 7713 CPU and 256GB RAM.

5.3 RQ1: Efficiency

As we mentioned in §4.3, to make it easier for users to conduct rewriting Wasm binaries, the semantics rewriter exposes a total of 31 APIs, which cover all the legal sections of Wasm. Therefore, a thorough evaluation on their efficiency is essential to evaluate BREWasm's usability. Based on each Wasm binary in Table 2, we invoke each of 31 APIs with proper arguments 1,000 times. For example, we call `insertGlobalVariable` to deliberately insert a global value at the beginning of the global section. Then, we record how long it will take to finish

Table 2. Representative Wasm binaries.

Name	Language	Type	Size (KB)
zigdom (B_1) [6]	C	web	3
stat (B_2) [55]	C	standalone	45
kindling (B_3) [40]	zig	web	3,373
rustexp (B_4) [38]	Rust	web	935
wasmnes (B_5) [46]	Rust	web	82
base64-cli (B_6) [53]	Rust	standalone	2,415
basic-triangle (B_7) [45]	Go	web	1,394
clock (B_8) [49]	Go	web	1,445
go-app (B_9) [11]	Go	web	4,302
audio (B_{10}) [50]	Go	web	8

the designated behavior, including all necessary stages, i.e., parsing, processing, fix-up, and encoding. The timing results are listed in Table 3. It is worth noting that in this experiment each API call requires a parsing process and an encoding process. However, under real scenarios, they are one-shot overhead to complete a given task through calling a series of APIs. Therefore, we list the parsing and encoding times in the second and the third row, respectively.

On average, it takes around 1.3s and 0.5s to parse and encode a Wasm binary, respectively. Although we can see that it takes 6.6s to parse B_9. However, this is because the Wasm binary consists of more than one million instructions, which will take up more than 100MB of space when converted to WAT format. Therefore, under normal scenarios, we can conclude that the parsing and encoding time will not exceed 2 to 3 seconds in total, which is acceptable as a one-shot overhead. Moreover, we can easily observe that each API takes only milliseconds or even less than a millisecond. Interestingly, among all these APIs, the operations related to *insertion* consume more time than other types of operations. This is because inserting an entry into a section requires fixing indices of subsequent entries to ensure continuity between indices. Also, the fixer under the semantics rewriter will have to enumerate all sections to identify if there are mismatched reference relations. Fortunately, these two fixing processes require no manual intervention.

Such a high efficiency is due to our engineering implementation as well as the time complexity. Specifically, four basic operations are defined in the section rewriter. As the `select` requires iterating on the given section, its time complexity is $O(n)$, where n refers to the length of the corresponding section. For the other three operations, their complexity are $O(1)$ because we adopt the hash table to store elements in sections. As for the semantics rewriter, the time complexity of APIs varies and depends on the implementation. Take the most extreme one, `insertHookFunction`, as an example. As we can see from Table 1, the nested loop causes its time complexity to be $O(n \cdot m)$, where n and m are proportional to the number of functions and their instructions. The time complexity of other APIs are typically $O(n)$.

Table 3. Consumed time (*ms*) of invoking APIs provided by the semantics rewriter, as well as the parsing and the encoding time on each Wasm binary.

	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}
parsing	6.46	55.66	173.71	1,023.41	119.09	725.94	2,174.82	1,794.90	6,651.99	2.94
encoding	2.58	21.28	79.29	408.89	42.32	332.95	617.66	586.59	3,294.77	2.00
Global Variable										
appendGlobalVariable	0.01	0.03	0.38	0.02	0.01	0.02	0.02	0.02	0.02	0.01
modifyGlobalVariable	0.06	1.08	1.34	24.14	1.83	48.43	37.07	37.57	123.13	0.04
deleteGlobalVariable	0.06	1.18	0.99	23.64	1.83	42.78	22.98	43.85	121.51	0.04
insertGlobalVariable	0.39	6.84	7.72	176.34	12.26	149.25	71.87	158.33	470.11	0.20
Import & Export										
insertImportFunction	0.31	4.61	4.99	76.62	9.67	101.00	48.13	102.51	300.64	0.16
appendImportFunction	0.08	1.30	1.59	29.48	2.11	38.29	15.62	38.51	118.36	0.05
modifyImportFunction	0.07	1.35	1.10	23.30	1.79	36.41	16.58	37.37	124.53	0.05
deleteImportFunction	0.07	2.52	1.58	36.25	1.77	36.98	16.21	36.67	120.52	0.04
insertExportFunction	0.07	1.64	1.12	23.31	1.85	37.22	16.51	36.84	119.84	0.04
appendExportFunction	0.07	1.89	2.20	23.93	1.91	34.72	16.56	42.74	113.94	0.04
modifyExportFunction	0.07	1.16	0.91	22.11	1.79	34.43	15.88	36.44	132.07	0.04
deleteExportFunction	0.07	1.29	1.85	26.81	2.13	35.22	14.97	33.75	120.78	0.04
Linear Memory										
appendLinearMemory	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.01
modifyLinearMemory	0.01	0.02	0.01	0.03	0.02	0.02	0.03	0.02	0.02	0.01
Function										
insertInternalFunction	0.33	4.48	4.49	65.43	8.16	100.77	47.59	99.82	318.12	0.17
insertIndirectFunction	0.38	4.64	5.87	89.97	10.11	131.80	63.39	132.27	429.44	0.21
insertHookFunction	0.74	9.88	10.87	176.79	20.58	260.23	119.99	261.16	833.86	0.40
deleteFuncInstr	0.09	0.92	2.30	23.25	1.94	32.90	17.11	33.74	121.37	0.05
appendFuncInstrs	0.07	0.86	1.32	23.55	1.81	35.42	17.90	35.13	122.17	0.04
insertFuncInstrs	0.08	0.94	1.95	24.39	1.99	33.39	16.26	33.91	126.50	0.06
modifyFuncInstr	0.08	0.88	2.04	22.69	2.23	34.58	16.80	34.50	121.72	0.05
appendFuncLocal	0.08	0.91	1.23	23.50	2.87	33.95	23.15	33.66	118.26	0.06
Custom Content										
modifyFuncName	0.01	0.02	0.02	0.12	0.04	0.12	0.12	0.12	0.40	0.01
deleteFuncName	0.08	0.91	1.55	23.21	1.96	34.81	15.64	33.60	121.61	0.06
insertFuncName	0.07	0.84	1.77	23.66	1.98	33.73	15.75	34.73	115.16	0.05
modifyGlobalName	0.08	0.91	1.54	22.97	2.07	34.50	15.63	34.56	120.61	0.06
deleteGlobalName	0.07	0.84	1.14	25.27	1.84	34.71	16.14	33.63	114.15	0.05
insertGlobalName	0.08	0.94	1.05	23.53	2.81	36.37	16.06	33.16	120.71	0.06
insertDataName	0.07	0.92	1.20	26.87	1.98	70.41	14.94	35.27	119.47	0.05
modifyDataName	0.08	0.89	1.39	23.40	2.45	35.04	16.10	33.45	119.95	0.05
deleteDataName	0.07	0.86	3.44	23.03	2.21	35.92	15.63	34.45	126.69	0.05

RQ-1 Answer: BREWASM exposes 31 semantics APIs that can efficiently achieve the corresponding goals. Though the parsing and the encoding processes on a Wasm binary take around 1.8 seconds, it is acceptable as a one-shot overhead compared to the negligible time it takes to execute semantics APIs.

5.4 RQ2: Correctness & Effectiveness

Correctly and effectively achieving the corresponding goals through APIs plays a vital role for BREWASM. However, it is insufficient to require only the correctness of the implementation of APIs in the section rewriter, which is due to the section coupling problem (see **C3** in §3.2). To evaluate the effectiveness of APIs of the semantics rewriter, we pass each Wasm binary after invoking an API shown in the first column of Table 3 to *wasm-validate*, an official syntax validator. Then, we manually double checked all 310 (31 APIs * 10 Binaries) cases to make sure the results are inline with the original intents.

According to the results, on the one hand, all rewritten Wasm binaries pass the validation of wasm-validate, indicating valid syntax; on the other hand, all 31 APIs perform correctly in their corresponding functionalities. Moreover, all 31 APIs resolve the **C2** and **C3** correctly. For example, when the API `insertImportFunction` is invoked, BREWASM not only rewrites the type section (if necessary) and the import section according to the import function, but also identifies if any instructions are affected, e.g., indices of the callee of `call` instructions and the indirect function table in the element section. BREWASM will automatically fix them to keep original semantics intact. Of course, all these evaluated APIs are passed with valid arguments. If invalid arguments are passed, e.g., inserting a function with a nonexistent index, the underlying `select` will return an empty list, leading to returning `false` by the following `insert` operation (see the formal definitions of `select` and `insert` in §4.2).

RQ-2 Answer: Based on the results of the automated verification tool and manual checks, we can conclude that these semantic APIs perform correctly in both syntax and functionalities.

5.5 RQ3: Practicability

We next demonstrate the practicability of BREWASM by illustrating some real scenarios that require rewriting Wasm binaries to achieve designated goals.

Case I: Binary Instrumentation. Binary instrumentation can be used to collect various runtime information of a program. D. Lehmann et al. [25] have implemented a dynamic analysis framework, named Wasabi, whose core is a Wasm binary instrumentation module. It specifies instrumentation rules for instructions to obtain runtime information during execution. For example, against a `call`, Wasabi inserts two functions (imported through the import section) before and after it, respectively, to record necessary information. Through APIs provided by BREWASM, the equivalent functionality can be implemented easily, as shown in Listing 1.3.

As we can see, L1 defines an instruction, `i32.const 5`, and L2 and L3 retrieve the type of the callee, i.e., the function indexed by `5`. Then, L5 and L6 construct two function signatures according to the callee's type. L8 invokes an API, named `appendImportFunction`, to introduce a function, whose name is `call_pre` belonging to a module named `hooks`, into import section. The same operation is

done in L10. Then, at L13, we construct a series of instructions, where the original `call` is wrapped by the newly declared `call_pre` and `call_post`. At L16, through another API, the original instruction will be replaced. Consequently, BREWASM achieves equivalent binary instrumentation like what Wasabi does.

Due to the fixed pattern of instrumentation used in Wasabi, modifying the underlying code is unavoidable to achieve other specific binary instrumentation functionalities. In contrast, BREWASM provides a large number of instruction-related general rewriting APIs, making it more flexible and convenient to implement the required instrumentation functionalities. In addition, in §4.2, our abstraction reduces the complexity of Wasm binaries, lowering the bars and the learning costs of using BREWASM.

```
1  instrs = [Instruction("i32.const", 5)]
2  calleeTypeidx = select(Function(5, _)).typeIdx
3  calleeType = select(Type(calleeTypeidx, _, _))
4  # construct call_pre and call_post
5  callPreFunctype = Type(_, ["i32"] + calleeType.typeArg,
       calleeType.typeRet)
6  callPostFunctype = Type(_, calleeType.typeArg, calleeType.
       typeRet)
7  # insert declaration to import section
8  appendImportFunction("hooks", "call_pre", callPreFunctype)
9  callPreFuncidx = select(Import(_, _, _, _))[-1].funcIdx
10 appendImportFunction("hooks", "call_post",
       callPostFunctype)
11 callPostFuncidx = select(Import(_, _, _, _))[-1].funcIdx
12 # replace the original call instruction
13 instrs.extend([Instruction("call", callPreFuncidx),
14                 Instruction("call", 5),
15                 Instruction("call", callPostFuncidx)])
16 modifyFuncInstr(Instruction("call", 5), instrs)
```

Listing 1.3. Achieve binary instrumentation through APIs provided by BREWASM.

Case II: Software Hardening. Software hardening is to enhance the security and stability of a program by updating it or implementing additional security measures [44]. In Wasm, unmanaged data, like strings, is stored in the linear memory, organized as a stack, and managed by a global variable representing the stack pointer. Because little protection measures are designed and adopted, traditional attacks, e.g., stack overflow, can exploit Wasm binaries leading to out-of-bound read and write [43]. However, through BREWASM, developers can easily conduct software hardening. For example, `callee` is originally potential for buffer overflow. After the hardening, it is wrapped by `hook`, which inserts a stack canary and validates its integrity around the invocation to `callee`. Listing 1.4 illustrates how to implement this goal via BREWASM.

As we can see, at L4, we randomly generate a canary number. Instructions from L5 to L11, responsible for validating the integrity of canary, are proposed by Fuzzm [27]. Similarly, the `funcbody`, the body of the function `hook`, defined

from L12 to L30 is also proposed by Fuzzm. To be specific, L12 to L18 deploys the generated canary into the linear memory. From L19 to L22, it dynamically generates the correct number of `local.get` according to the signature of the callee, which is retrieved by the API `getFuncFunctype`. After the instruction `call $callee` (L24), we insert the already defined canary validation piece as the operand of a `block` instruction (L25). If the value of canary is unchanged, indicating no buffer overflow, the instructions from L26 to L30 will restore the stack and give the control back to `caller`.

```
1  # suppose calleeFuncIdx is given by the user
2  # the first global is a stack pointer
3  stackPointerIdx = 0
4  canary = random.randint(1, 10000)
5  canaryValidateInstrs = [
6              Instruction("global.get", stackPointerIdx),
7              Instruction("i64.load"),
8              Instruction("i64.const", canary),
9              Instruction("i64.eq"),
10             Instruction("br_if", 0),
11             Instruction("unreachable")]
12 funcbody = [Instruction("global.get", stackPointerIdx),
13             Instruction("i32.const", 16),
14             Instruction("i32.sub"),
15             Instruction("global.set", stackPointerIdx),
16             Instruction("global.get", stackPointerIdx),
17             Instruction("i64.const", canary),
18             Instruction("i64.store", canary)]
19 calleeTypeidx = select(Function(calleeFuncIdx, _)).typeIdx
20 calleeType = select(Type(calleeTypeidx, _, _))
21 for idx, _ in enumerate(calleeType.paramsType):
22     funcbody.append(Instruction("local.get", idx))
23 funcbody.extend([
24             Instruction("call", calleeFuncIdx),
25             Instruction("block", canaryValidateInstrs),
26             Instruction("global.get", stackPointerIdx),
27             Instruction("i32.const", 16),
28             Instruction("i32.add"),
29             Instruction("global.set", stackPointerIdx),
30             Instruction("return")])
31 insertHookFunction(calleeFuncIdx, calleeType.paramsType,
       calleeType.resultsType, funcbody, locals = [])
```

Listing 1.4. Achieve software hardening through APIs provided by BREWasm.

In Fuzzm, the stack canary protection is implemented in a similar way. However, since a function may have multiple exit points, Fuzzm modifies the control flow of the function to have only one exit point. This might disrupt the original semantics of the function. In contrast, BREWasm's `insertHookFunction` provides a better solution for this issue, as it minimizes the modifications made to the original function's instructions in the Wasm binary.

Case III: Fuzzing. Fuzzing is an automated software testing technique that can discover security and stability issues by using random files as input [34]. One of the widely adopted approaches to generate random files is *mutation-based*, i.e., generating new files by mutating existing files. Wasm-mutate can mutate the given Wasm binary while keeping semantic equivalence. One of its mutation strategies is *module structure mutation*, e.g., introducing an isolated function. This can also be easily done by BREWasm, as shown in Listing 1.5.

```
1  typeSecLen = len(select(Type(_, _, _)))
2  functype = select(Type(random.randint(0, typeSecLen), _, _
       ))
3  # construct a function according to a random signature
4  funcbody = []
5  for retType in functype.resultsType:
6      if retType == "i32":
7          funcbody.append(Instruction("i32.const", 0))
8      elif retType == "i64":
9          funcbody.append(Instruction("i64.const", 0))
10     elif retType == "f32":
11         funcbody.append(Instruction("f32.const", 0.0))
12     elif retType == "f64":
13         funcbody.append(Instruction("f64.const", 0.0))
14 # randomly insert the function
15 importFuncNum = len(select(Import(_, _, _, _)))
16 internalFuncNum = len(select(Function(_, _)))
17 funcNum = importFuncNum + internalFuncNum
18 insertInternalFunction(random.randint(importFuncNum,
       funcNum), functype.paramType, functype.resultType,
       funcbody, locals = [])
```

Listing 1.5. Achieve add function mutation through APIs provided by BREWasm.

As we can see, L1 and L2 randomly select a type serving as a function signature. According to the designated signature, L4 to L13 construct the body of the function by inserting meaningless instructions to ensure stack balance. Finally, at L18, the generated function will be randomly inserted into the Wasm binary, and the original semantic keeps intact.

There is no rewriting component in Wasm-mutate, thus achieving this goal needs to decode and encode the relevant sections. In contrast, BREWasm parses the entire Wasm binary into an object, which simplifies many extra operations. Additionally, since BREWasm implements the semantics rewriter, inserting a function only requires invoking a single API, without the need to rewrite the relevant sections in turn as in Wasm-mutate.

Other Application Scenarios of BREWasm. Except for the three concrete applications we mentioned above, BREWasm can also be applied in other scenarios, as summarized in Table 4.

Code Obfuscation. Some traditional code obfuscation methods can be implemented. For instance, users can insert global variables or internal functions as

Table 4. Other application scenarios of BREWasm.

Scenario	Applications	Related Semantics APIs

Code Obfuscation	Opaque predicates obfuscation;
	Memory encryption;
	Debug info obfuscation
		`insertGlobalVariable`
		`insertInternalFunction`
		`modifyLinearMemory`
		`modifyFuncName`

Software Testing	Runtime testing; WASI function testing	`modifyFuncInstr` `appendImportFunction`
Program Repair	Bug fixing	`modifyFuncInstr` `appendFuncLocal`
Software optimization	Instruction optimization	`modifyFuncInstr` `appendFuncLocal`

opaque predicates [12] into a Wasm binary. Moreover, users can change the debug information in the custom section, like obfuscating readable names of functions, to make the Wasm binary unreadable for attackers.

Software Testing. The exposed APIs can also be used in software testing. For example, Y. Zhang et al. [62] have proposed a method to mutate an instruction's operands constantly to examine if the instruction follows the specification. Through `modifyFuncInstr`, users can easily achieve operands rewriting. Users can even call `appendExportFunc` to export the result of the instruction to alleviate the workload of results comparing. The same approach can be applied in testing imported functions as well.

Program Repairing. BREWasm can be used to fix bugs in Wasm binaries without source code. For example, BREWasm can insert a wrapper function around addition instructions. Within the wrapper, as integer overflow can be detected easily, users can choose to either correct results or raise an exception.

Software Optimization. Instructions optimization can be easily achieved. For example, a piece of Wasm bytecode can be optimized with a higher-level optimization during the compilation, which, however, requires accessing the source code. In contrast, through the APIs offered by BREWasm, users can easily match a piece of code with a pre-defined pattern. Then, through `modifyFuncInstr` and `appendFuncLocal`, the code snippet can be updated to an optimized one.

RQ-3 Answer: Our exploration suggests that it is practical to achieve various kinds of complicated Wasm binary rewriting tasks by combing the APIs provided by BREWasm. Comparing with the cumbersome implementation of the specific tasks, the work built on BREWasm is effortless and user-friendly.

6 Related Work

Wasm Binary Analysis. As an emerging stack-based language, WebAssembly can be applied inside or outside the browser [2]. Lots of work focused on Wasm binary analysis [24,21,26,35,42,19,20]. For example, Lehmann and Pradel [24,21] pay attention to the memory issues in Wasm, e.g., exploitable buffer overflow.

They found that some memory issues in the source code will still be exploitable in compiled Wasm binaries. In addition, Quentin Stiévenart et al. [42] overcame the challenges of dependency analysis at the binary level and presented an approach to statically slice Wasm programs for the following analysis.

Static Binary Rewriting. Researchers have proposed lots of static rewriting tools against native programs [33,16,13,59,30]. Some of them, e.g., Alto [33], SASI [16] and Diablo [13], can only rewrite programs with the assistance of debug information or the ones compiled by a specific compiler. Recently, E9Patch [14] proposes a control-flow-agnostic rewriting method that inserts jumps to trampolines without the need to move other instructions, which significantly improves the scalability and robustness. The Bytecode Alliance provides two tools, wasm-parser [51] and wasm-encoder [51], for parsing and encoding Wasm binaries, while using them would face the same challenge of Wasm binary complexity. As a comparison, our proposed abstraction model in §4.1 could reduce the complexity of Wasm binaries when implementing the section rewriting. To the best of our knowledge, there is no static binary rewriting framework for Wasm yet.

7 Limitations & Discussion

Structured Control Flow Rewriting. Wasm adopts a special and complicated control flow structure, named *structured control flow* [3]. Specifically, a Wasm function is composed of a set of sequential or nested code blocks, each of which has to be led by a `block` or `loop` instruction. Moreover, some instructions can guide the control flow to the destination code blocks, like `end` and `br`. In other words, by rewriting such instructions, BREWasm can handle the control flow of Wasm binaries to some extent. However, to guarantee the semantic consistency, users cannot simply change a `block` into a `loop` (which turns a normal code block into a loop structure), but have to implement their own APIs with four operations in the section rewriter we proposed, which is tedious, error-prone, and case-specific. No existing work is able to rewrite the structured control flow in a flexible and general way, which will be one of our future work.

Concerns about Reinventing Wheels. Some existing tools have similar design purpose or functionalities with BREWasm, thus there may have concerns about reinventing wheels. We underline that all components in BREWasm are irreplaceable. Specifically, wasm-parser and wasm-encoder are implemented by the Bytecode Alliance official, however, directly adopting them will overwhelm users. Though they can parse and encode the given Wasm binary and fully support all sections defined in the specification, parsing and encoding require approximately 1,300 and 800 lines of code, respectively. It is tedious and infeasible to compose a script with more than 2K LOC to conduct binary rewriting. In addition, though Wasabi has implemented a wasabi_wasm module, which is able to conduct static instrumentation, it is insufficient in binary rewriting. On the one hand, it is designed specifically for binary instrumentation, indicating that it cannot implement removing or updating vectors in sections, which will significantly hinder its ability in terms of binary rewriting. On the other hand, it

does not consider the section coupling challenge as we mentioned in **C3**. Moreover, some sections are not supported by its module, like the elem section. The lack of ability on rewriting some sections will dramatically impact the flexibility of rewriting. Consequently, implementing BREWasm instead of combining existing modules or tools does not indicate reinventing wheels.

8 Conclusion

We present BREWasm, a general binary rewriting framework for WebAssembly. BREWasm can properly handle inherent challenges of Wasm binary rewriting, including the highly complicated binary structure, strict static syntax verification, and coupling among sections. Based on representative Wasm binaries, BREWasm illustrates its efficiency and effectiveness in rewriting process. Through three cases inspired by existing work and real-world scenarios, BREWasm also proves its practicality and usability.

Acknowledgement

We have great thanks to all anonymous reviewers and our shepherd, Prof. Jingling Xue. This work was supported in part by National Key R&D Program of China (2021YFB2701000), the National Natural Science Foundation of China (grants No.62072046 and 62141208), and Xiaomi Young Talents Program.

References

1. Leb128 algorithm (Feb 2023), https://en.wikipedia.org/wiki/LEB128
2. official webpage (Feb 2023), https://webassembly.org/docs/use-cases/
3. structured control flow (Feb 2023), https://tinygo.org/docs/guides/webassembly/
4. Alliance, B.: Github wasm-tools repository (Feb 2023), https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate
5. Becker, M., Baldin, D., Kuznik, C., Joy, M.M., Xie, T., Mueller, W.: Xemu: an efficient qemu based binary mutation testing framework for embedded software. In: Proceedings of the tenth ACM international conference on Embedded software. pp. 33–42 (2012)
6. Bhattarai, S.: Github zig-wasm-dom repository (Feb 2023), https://shritesh.github.io/zig-wasm-dom/
7. Brito, T., Lopes, P., Santos, N., Santos, J.F.: Wasmati: An efficient static vulnerability scanner for webassembly. Computers & Security **118**, 102745 (2022)
8. Bruening, D., Amarasinghe, S.: Efficient, transparent, and comprehensive runtime code manipulation. Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering ... (2004)
9. Cabrera Arteaga, J., Donde, S., Gu, J., Floros, O., Satabin, L., Baudry, B., Monperrus, M.: Superoptimization of webassembly bytecode. In: Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming. pp. 36–40 (2020)

10. Cabrera-Arteaga, J., Monperrus, M., Toady, T., Baudry, B.: Webassembly diversification for malware evasion. arXiv preprint arXiv:2212.08427 (2022)
11. Charriere, M.: LOFIMUSIC website (Feb 2023), `https://lofimusic.app/collegemusic-lonely`
12. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 184–196 (1998)
13. De Sutter, B., De Bus, B., De Bosschere, K.: Link-time binary rewriting techniques for program compaction. ACM Transactions on Programming Languages and Systems (TOPLAS) **27**(5), 882–945 (2005)
14. Duck, G.J., Gao, X., Roychoudhury, A.: Binary rewriting without control flow recovery. In: Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation. pp. 151–163 (2020)
15. eosio: eosio official website (Feb 2023), `https://eos.io/`
16. Erlingsson, U., Schneider, F.B.: Sasi enforcement of security policies: A retrospective. In: Proceedings of the 1999 workshop on New security paradigms. pp. 87–95 (1999)
17. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with webassembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 185–200 (2017)
18. Hall, A., Ramachandran, U.: An execution model for serverless functions at the edge. In: Proceedings of the International Conference on Internet of Things Design and Implementation. pp. 225–236 (2019)
19. He, N., Zhang, R., Wang, H., Wu, L., Luo, X., Guo, Y., Yu, T., Jiang, X.: Eosafe: Security analysis of eosio smart contracts. In: USENIX Security Symposium. pp. 1271–1288 (2021)
20. He, N., Zhao, Z., Wang, J., Hu, Y., Guo, S., Wang, H., Liang, G., Li, D., Chen, X., Guo, Y.: Eunomia: Enabling user-specified fine-grained search in symbolically executing webassembly binaries. arXiv preprint arXiv:2304.07204 (2023)
21. Hilbig, A., Lehmann, D., Pradel, M.: An empirical study of real-world webassembly binaries: Security, languages, use cases. In: Proceedings of the Web Conference 2021. pp. 2696–2708 (2021)
22. Hundt, R.: Hp caliper: A framework for performance analysis tools. IEEE Concurrency **8**(4), 64–71 (2000)
23. Kim, T., Kim, C.H., Choi, H., Kwon, Y., Saltaformaggio, B., Zhang, X., Xu, D.: Revarm: A platform-agnostic arm binary rewriter for security applications. In: Proceedings of the 33rd Annual Computer Security Applications Conference. pp. 412–424 (2017)
24. Lehmann, D., Kinder, J., Pradel, M.: Everything old is new again: Binary security of webassembly. In: Proceedings of the 29th USENIX Conference on Security Symposium. pp. 217–234 (2020)
25. Lehmann, D., Pradel, M.: Wasabi: A framework for dynamically analyzing webassembly. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 1045–1058 (2019)
26. Lehmann, D., Pradel, M.: Finding the dwarf: Recovering pecise types from webassembly binaries. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 410–425 (2022)

27. Lehmann, D., Torp, M.T., Pradel, M.: Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly (10 2021), https://arxiv.org/pdf/2110.15433.pdf
28. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices **40**(6), 190–200 (2005)
29. Mäkitalo, N., Mikkonen, T., Pautasso, C., Bankowski, V., Daubaris, P., Mikkola, R., Beletski, O.: Webassembly modules as lightweight containers for liquid iot applications. In: Web Engineering: 21st International Conference, ICWE 2021, Biarritz, France, May 18–21, 2021, Proceedings. pp. 328–336. Springer (2021)
30. McSema: Github McSema repository (Feb 2023), https://github.com/lifting-bits/mcsema
31. MDN: MDN web docs website (Feb 2023), https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_wasm
32. Musch, M., Wressnegger, C., Johns, M., Rieck, K.: New kid on the web: A study on the prevalence of webassembly in the wild. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16. pp. 23–42. Springer (2019)
33. Muth, R., Debray, S.K., Watterson, S., De Bosschere, K.: alto: a link-time optimizer for the compaq alpha. Software: Practice and Experience **31**(1), 67–101 (2001)
34. Nagy, S., Nguyen-Tuong, A., Hiser, J.D., Davidson, J.W., Hicks, M.: Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In: 30th USENIX Security Symposium (2021)
35. Naseem, F.N., Aris, A., Babun, L., Tekiner, E., Uluagac, A.S.: Minos: A lightweight real-time cryptojacking detection system. In: NDSS (2021)
36. Nieke, M., Almstedt, L., Kapitza, R.: Edgedancer: Secure mobile webassembly services on the edge. In: Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking. pp. 13–18 (2021)
37. Payer, M., Barresi, A., Gross, T.R.: Fine-grained control-flow integrity through binary hardening. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings 12. pp. 144–164. Springer (2015)
38. Pilfold, L.: Rustexp website (Feb 2023), https://rustexp.lpil.uk/
39. PyPI: PyPI cyleb128 library (Feb 2023), https://pypi.org/project/cyleb128/
40. Shenton, C.: Github kingling repository (Feb 2023), https://github.com/cshenton/kindling
41. Srivastava, A., Eustace, A.: Atom: A system for building customized program analysis tools. In: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation. pp. 196–205 (1994)
42. Stiévenart, Q., Binkley, D.W., De Roover, C.: Static stack-preserving intra-procedural slicing of webassembly binaries. In: Proceedings of the 44th International Conference on Software Engineering. pp. 2031–2042 (2022)
43. Stiévenart, Q., De Roover, C., Ghafari, M.: Security risks of porting c programs to webassembly. In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. pp. 1713–1722 (2022)
44. Strackx, R., Piessens, F.: Fides: Selectively hardening software application components against kernel-level or process-level malware. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 2–13 (2012)
45. Suedmeier, E.: wasm-basic-triangle website (Feb 2023), https://shritesh.github.io/zig-wasm-dom/

46. Takahiro: nes-rust-ecsy website (Feb 2023), https://takahirox.github.io/nes-rust-ecsy/index.html
47. Tian, L., Shi, Y., Chen, L., Yang, Y., Shi, G.: Gadgets splicing: dynamic binary transformation for precise rewriting. In: 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 155–167. IEEE (2022)
48. TinyGo: TinyGo official docs webpage (Feb 2023), https://tinygo.org/docs/guides/webassembly/
49. Ts, J.: Github clockexample-go-webassembly repository (Feb 2023), https://github.com/Yaoir/ClockExample-Go-WebAssembly
50. Turner, A.: Github wasm-by-example repository (Feb 2023), https://github.com/torch2424/wasm-by-example/tree/master/examples/reading-and-writing-audio/demo/go
51. wabt: wabt tool website (Feb 2023), https://github.com/WebAssembly/wabt
52. Wang, W., Ferrell, B., Xu, X., Hamlen, K.W., Hao, S.: Seismic: Secure in-lined script monitors for interrupting cryptojacks. In: Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II 23. pp. 122–142. Springer (2018)
53. wapm: base64-cli app in wapm (Feb 2023), https://takahirox.github.io/nes-rust-ecsy/index.html
54. wasabi: Github wasabi repository (Feb 2023), https://github.com/danleh/wasabi
55. WAVM: Github wavm repository (Feb 2023), https://github.com/WAVM/WAVM/tree/master/Test/wasi
56. WebAssembly: WebAssembly specification webpage (Feb 2023), https://webassembly.github.io/spec/core/binary/index.html
57. WebAssembly: WebAssembly static validation algorithm (Feb 2023), https://webassembly.github.io/spec/core/appendix/algorithm.html
58. WebAssembly: WebAssembly website (Feb 2023), https://webassembly.org/
59. Williams-King, D., Kobayashi, H., Williams-King, K., Patterson, G., Spano, F., Wu, Y.J., Yang, J., Kemerlis, V.P.: Egalito: Layout-agnostic binary recompilation. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 133–147 (2020)
60. Xu, Y., Xu, Z., Chen, B., Song, F., Liu, Y., Liu, T.: Patch based vulnerability matching for binary programs. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 376–387 (2020)
61. Zakai, A.: Emscripten: an llvm-to-javascript compiler. In: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. pp. 301–312 (2011)
62. Zhang, Y., Cao, S., Wang, H., Chen, Z., Luo, X., Mu, D., Ma, Y., Huang, G., Liu, X.: Characterizing and detecting webassembly runtime bugs. arXiv preprint arXiv:2301.12102 (2023)