Electronic Preprint for Journal of Information Processing Vol.28

Regular Paper

Finding Errors in Registrations of Local Variables Using Coccinelle for Accurate Garbage Collection

Tomoharu Ugawa[1,a)] Taiki Fujimoto[1,†1]

Received: June 26, 2019, Accepted: September 24, 2019

Abstract: For accurate garbage collection (GC), all pointers belonging to the root set must be found. In a virtual machine (VM) implemented in C language, local variables of C language may contain pointers. Thus, some VMs add the values or addresses of local variables to a table that is visible to GC. However, this approach is error-prone because it requires adding local variables and removing them correctly though the entire source code of the VM. In this research, we checked if local variables are added and removed correctly by pattern matching against control flow graphs of the source code of the VM. We applied this check to the VM of a subset of JavaScript we are developing and found that it could identify many cases of missed adding and redundant adding.

Keywords: garbage collection, bug finding, program analysis

1. Introduction

In moving garbage collection (GC), such as compaction and copying GC, pointers are fixed up when objects are moved so that all the pointers point to new locations. Thus, it is crucial that moving GC knows the addresses of all the GC roots accurately. In the case of interpreters and virtual machines (VMs) implemented in C, the root set includes local variables that point to managed data in the heap.

However, it is not easy to accurately enumerate all local variables that have pointers to the heap because GC does not know the locations of these variables, which are placed on the execution stack together with other data, or if the value of a local variable is a pointer or not. Thus, VM developers usually either develop a supporting mechanism in the compiler, such as stack maps, or add local variables to the root set and remove them explicitly.

We are currently developing eJSVM [1], [2], a JavaScript VM for embedded systems. Although mark-sweep GC, which does not move objects, is used in the current version of eJSVM, we plan to implement a moving GC in a future version. To the end, we add the addresses of local variables holding pointers to the root set and remove them explicitly. An example is shown in **Fig. 1**, which depicts the source code of eJSVM, modified for explanation. `GC_PUSH` is a macro to add the given variable to the root set, and `GC_POP` is a macro to remove. The variables of the JSValue type contain pointers to the heap. This code adds all the JSValue type local variables, including formal parameters, at the entry of the function, and removes them before leaving the function. Note that it removes before the `return` statement in the middle of the function, as well.

Maintaining the root set with `GC_PUSH`es and `GC_POP`s introduces unusual discipline to the programming, and thus it is error-prone. The risk of error can increase for three reasons:

- Programmers tend to omit code that they feel is unnecessary. Moreover, in our development of eJSVM, we are motivated to reduce the VM footprint. For example, we inserted `GC_PUSH`es and `GC_POP`s only to the lines shown in **Fig. 2** in reality (see Section 2.3 for detail).
- Programmers have to maintain `GC_PUSH`es and `GC_POP`s even when they modify programs.
- In eJSVM, VM users, as well as VM developers, may develop built-in functions in C to access their hardware.

Furthermore, the bug of missing `GC_PUSH`es and `GC_POP`s is difficult to find and fix by testing because errors due to this kind of bug are rarely reproduced.

We explored various methods that check if `GC_PUSH`es and `GC_POP`s are inserted correctly and found that Coccinelle [3], [4], a tool for pattern matching against control flow graphs, could detect missing `GC_PUSH`es with high accuracy. Coccinelle receives the pattern of a control flow graph written in a domain specific language (DSL) and then uses it to performs pattern matching.

```
1  get_object_prop
2  (Context *ctx, JSValue o, JSValue p) {
3    JSValue ret = NULL;
4    GC_PUSH(o);GC_PUSH(p);GC_PUSH(ret);
5    if (!is_string(p))
6      p = to_string(ctx, p);
7    do {
8      if (get_prop(o, p, &ret) == SUCCESS) {
9        GC_POP(ret);GC_POP(p);GC_POP(o);
10       return ret;
11     }
12   } while (get___proto__(o, &o) == SUCCESS);
13   GC_POP(ret);GC_POP(p);GC_POP(o);
14   return JS_UNDEFINED;
15 }
```

Fig. 1 Source code of eJSVM, where `GC_PUSH`es and `GC_POP`s are inserted straightforwardly.

[1] Kochi University of Technology, Kami, Kochi 782–8502, Japan
[†1] Presently with TOSCO Corporation
[a)] ugawa.tomoharu@kochi-tech.ac.jp

```
 1 get_object_prop
 2 (Context *ctx, JSValue o, JSValue p) {
 3   JSValue ret;
 4   if (!is_string(p)) {
 5     GC_PUSH(o);
 6     p = to_string(ctx, p);
 7     GC_POP(o);
 8   }
 9   do {
10     if (get_prop(o, p, &ret) == SUCCESS)
11       return ret;
12   } while (get___proto__(o, &o) == SUCCESS);
13   return JS_UNDEFINED;
14 }
```

Fig. 2 Source code of eJSVM.

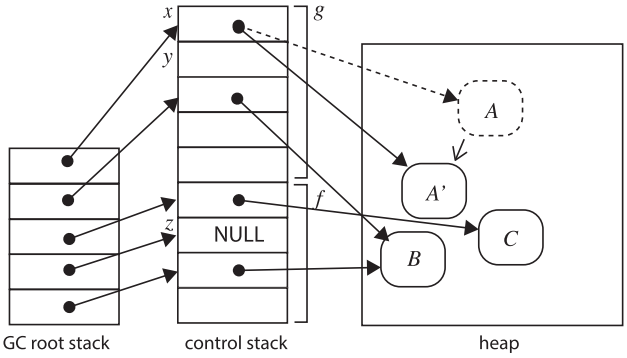

Fig. 3 GC root stack.

It is currently used for refactoring and bug finding in the Linux kernel project.

To evaluate the accuracy of this pattern matching approach, we removed GC_PUSHes and GC_POPs from the source code of eJSVM and then tried to see if Coccinelle could detect them. Results showed that Coccinelle with our pattern was able to detect all of them, excluding those that were redundant. Coccinelle also reported that more GC_PUSHes were missing, and we confirmed that all of them were necessary except for a single case.

This paper reports our experience with debugging, where we used Coccinelle to find the bugs of missing GC_PUSHes and GC_POPs of local variables. This paper also reports the case study where we applied the same method to the source code of another project.

2. Maintaining Root Set in eJSVM

2.1 Overview

eJSVM uses a stack (GC root stack) as the root set that stores the addresses of local variables. **Figure 3** shows the control stack and GC root stack, where a function f called g. Local variables are allocated in function frames, which are located on the control stack. Thus, it is reasonable to use a stack to implement the root set as well.

When an object A is moved to A' by GC, GC updates the pointers to A (x in the figure) with A's new location A'. To enable GC to do this, the addresses of local variables, rather than their values, are pushed to the GC root stack.

GC dereferences the pointers stored in the local variables whose addresses are stored in the GC root stack. Thus, uninitialized variables should not be pushed to the GC root stack. For example, even if we know that variable y will eventually have a pointer, if we push the address of y before y is initialized, GC would dereference the value that happened to be stored in y when GC is invoked. GC does not, however, dereference the values of the variables if they are initialized with NULL, as variable z is.

The program of eJSVM pushes and pops the addresses of local variables using the GC_PUSH and GC_POP macros. eJSVM has a build option to enable an extra check to ensure GC_POPs correspond to GC_PUSHes. For this built option, GC_POPs requires a variable name, which is only used in a debug build.

2.2 Variables Containing Pointers to the Heap

In eJSVM, the types of variables determine if their values are pointers to the heap or not. More specifically, only variables of the JSValue type, which is for values of JavaScript, and pointer types to some VM internal data structures, such as hidden classes [5], have pointers to the heap. Although a JSValue type variable may have a value other than a pointer, such as an integer, it always has a tag in its least significant bits to distinguish if it is a pointer or not. As for VM internal data structures, their types determine if they are allocated in the heap or elsewhere – allocated statically for example. In our investigation using Coccinelle, we deal with the JSValue type and pointer types to the data structures in the heap in exactly the same way. In the rest of this paper, we refer to both of them as JSValue types.

2.3 Removing Redundant GC_PUSH and GC_POP Insertion

The program shown in Fig. 1 GC_PUSHes all of the three JSValue type variables at the entry of the function and GC_POPs all of them at both exits. In addition, `ret`, which is declared in the function, is initialized with NULL before it is GC_PUSHed. Although this works correctly, some GC_PUSHes and GC_POPs are redundant. This redundancy not only applies overhead to the execution time but also increases the VM footprint. Since eJSVM is used for embedded systems, we want to avoid this redundancy.

Because JavaScript programs are single-threaded, GC is not invoked asynchronously by other threads, unlike in multi-threaded programs. Furthermore, in eJSVM, the VM context is required to invoke GC. Thus, the positions of code where GC may be invoked are limited to function calling sites, where a pointer to a `Context` type structure is passed as an argument. Since GC cares about the accuracy of the root set only when GC is invoked, we can remove GC_PUSHes and GC_POPs on the basis of the live ranges of variables so that all the addresses of live variables are stored in the GC root stack whenever GC is invoked.

For example, in Fig. 1, the only position of code that may cause GC is the call of `to_string` on line 6, and the only variable that has a live range covering this position is `o`. As for `ret`, GC_PUSH is not necessary because a value is stored in `ret` for the first time on line 8. Note that, on line 8, a value is stored in `ret` through its address passed to `get_prop`.

With `p`, matters are more complicated. A value is passed as an argument to `p`, and `p` is used on line 8. However, because line 6 writes the return value of `to_string` to `p`, it splits `p`'s live range. Even if line 6 causes GC, and if GC moves the object passed as an argument to `p`, `p` will be overwritten with the return value of `to_string`. Thus, no GC_PUSH is required for `p`.

```
1  JSValue e;
2  ...
3  e = args[i];
4  ...
5  while (k < len) {
6    if (has_array_element(e, k)) {
7      subElement =
8        get_array_prop(ctx, e,
9                       cint_to_fixnum(k));
10     set_array_prop(ctx, a, cint_to_fixnum(n),
11                    subElement);
12   }
13   n++;
14   k++;
15 }
```

Fig. 4 Example of program needed to consider the execution path.

Furthermore, o does not necessarily have to be GC_PUSHed as early as at the function entry. line 6, calling to_string, is executed only when p does not have a string value. If we GC_PUSH and GC_POP in the then-clause of this if statement, we can deduplicate GC_POPs. This is also likely to improve the performance because the condition of the if statement is unlikely to be satisfied. This is because the get_object_prop function defined in Fig. 1 is to obtain the property of an object, and the property name passed to p is likely to be a string value.

A program without redundant insertion of GC_PUSHes and GC_POPs is shown in Fig. 2.

2.4 Inserting GC_PUSH While Considering Execution Path

We have to consider execution paths to judge if GC_PUSH is necessary or not. **Figure 4** shows the implementation of built-in function Array.prototype.concat. In this implementation, e, a JSValue type variable, has to be GC_PUSHed before the function call on line 8, which may invoke GC, because it is used on line 6. Although it is used above the call site in the program text, e may be used after the function call because they are both in a while statement.

3. Coccinelle

We used Coccinelle [3], [4] for pattern matching against the execution paths of programs. Coccinelle receives rules written in a DSL, SmPL, and matches them against the control flow graph of a program. As the rules for Coccinelle are called semantic patches, Coccinelle can also rewrite the matched piece of code. This is used for refactoring and bug finding in the Linux kernel project.

In SmPL, we can use fragments of C program and wildcards such as "..." as patterns to describe rules. Coccinelle converts these patterns into computational tree logic expressions extended with meta-variables (CTL-V expressions), and then performs model checking against the Kripke structure of the control flow graph. A control flow graph is regarded as a Kripke structure, where states are statements of the program, and atomic properties are whether or not each statement satisfies the conditions written in the rule. As a condition of statements, we can specify a statement that may contain meta-variables, or expressions that the statement should have. Because Coccinelle utilizes CLT, we can quantify a wildcard with exists or forall.

A single SmPL file may contain multiple rules. Rules can use meta-variables to which match results are bound in their preceding rules. This allows us to add conditions to a rule and remove false positives.

```
@MissingPush depends on use && !falseuse && !immass@
identifier pre.push, decl.v;
expression e, gc.gc_fun;
position gc.gc_p, decl.decl_p;
type decl.T;
@@
(
 T v@decl_p;
|
 T v@decl_p = e;
)
... when != push(&v)
    when exists
gc_fun@gc_p
```

Fig. 5 Rule to find missing GC_PUSH.

```
1 JSValue x = NULL;    /*T v@decl_p = e; matches*/
2 if (flag == 1)
3   cause_gc(context);/*gc_fun@gc_p matches*/
```

Fig. 6 Program that causes GC on one path but not the other.

Here, we briefly explain the grammar of SmPL using the rule shown in **Fig. 5** as an example. This rule finds missing GC_PUSHes (we explain how this rule works in Section 4). The first line contains the name of this rule, MissingPush, and the names of depending rules, use, falseuse, and immass. With this dependency description, this pattern is used for the combinations of meta-variables that match use but do not match falseuse or immass.

The following lines up to @@ define meta-variables. Descriptions such as decl.v allow rules to use the meta-variables of preceding rules. Not only the syntactic element type meta-variables but also the meta-variables of the position type, which are bound to the positions of a program where the statements occur, are available.

The lines below @@ define the body of the pattern. The pattern element "(|)" describes a choice, and "..." describes that there is an arbitrary number of statements [*1]. A wildcard "..." may come with conditions, e.g., "when != push(&v)" describes that there are no statements having push(&v), and "when exists" describes that there exists some execution path that satisfies the following pattern [*2]. Note that exists does not describe the existence of statements corresponding to "..." but rather the existence of execution paths to an exits of the function that the following pattern matches. For example, the rule shown in Fig. 5 matches the program shown in **Fig. 6**, which has two execution paths, and gc_fun@gc_p matches one of them, because exists describes that gc_fun@gc_p matches at least one of the execution paths. If there were no exists, this rule would not match the program shown in Fig. 6.

A syntactic element may be attached with a meta-variable describing the position of an occurrence, such as @decl_p. For

[*1] The following statement is connected with the X (next) modal operator if there is no "...", and it is connected with the U (until) modal operator if there is.

[*2] Translated into the EU modal operator.

example, if we take the meta-variable declaration into account, gc_fun@gc_p represents statements that call some function bound to gc_fun that was found in the gc rule, and the call site is at some position that is found in the gc rule. Note that an expression followed by a semi-colon represents a statement while an expression without a semi-colon represents statements containing the expression.

Coccinelle allows us to describe a part of a rule using OCaml and Python. In this research, we describe conditions on meta-variables in Python. We also use Python in the body of some rules to print messages to the terminal when Coccinelle finds bugs.

4. Rule to Find Bugs

We developed rules for Coccinelle to find the following bugs:

- missing GC_PUSH
- storing address of JSValue type variable to variables
- double GC_PUSH
- missing GC_POP
- GC_POP before GC_PUSH
- premature GC_POP
- double GC_POP
- GC_PUSHing wrong type of variable

Storing an address of a JSValue type variable x to a variable p is not itself a bug. However, the stored address may be used after x is GC_POPed. We consider this case to be a bug. We conservatively forbid stores of addresses of JSValue type variables because the current version of eJSVM does not do so, and it is unlikely to cause problems in the future. "Premature GC_POP" is a bug that GC_POPs a variable that may be used after GC following to the GC_POP.

In the rest of this section, we focus on the rule to find the missing GC_PUSH bug, which is the most complicated. We developed the other rules in the same way. Furthermore, we focus on the case where the variable to be GC_PUSHed is a local variable. The rules to handle the cases where the variable is a local variable and a parameter are essentially the same, but developed as separate rules.

4.1 Structure of Missing GC_PUSH Bug

The program shown in Fig. 4 and the program obtained from the one shown in Fig. 2 by removing GC_PUSHes are examples that the rule to find the missing GC_PUSH bug should find. We first enumerate the conditions under which GC_PUSH is required.

- **gc**: A function that may cause GC is called. We refer to the function as *gc_fun* and the position of the call site as *gc_p*.
- **decl**: A JSValue type variable is declared. We refer to the variable as v and the position of the declaration as *decl_p*.
- **use**: Variable v is used after *gc_p*. We refer to the position of such use of v as *use_p*.

If all of these three conditions are satisfied, a GC_PUSH of v may be required between *decl_p* and *gc_p*. For example, in the program shown in Fig. 4, lines 8 and 10 satisfy the **gc** condition, and line 1 satisfies the **decl** condition. Whichever line 8 or 10 *gc_p* is, line 6 satisfies the **use** condition. Thus, the program satisfies all conditions, where *decl_p* is line 1 and *gc_p* is line 8 or 10. In fact, this program requires GC_PUSH before line 8 is executed.

```
1    JSValue x, y;
2    y = cause_gc(ctx);
3    x = arg[1];
4    return x == y;
```

Fig. 7 Program that has assignment to JSValue-type variable.

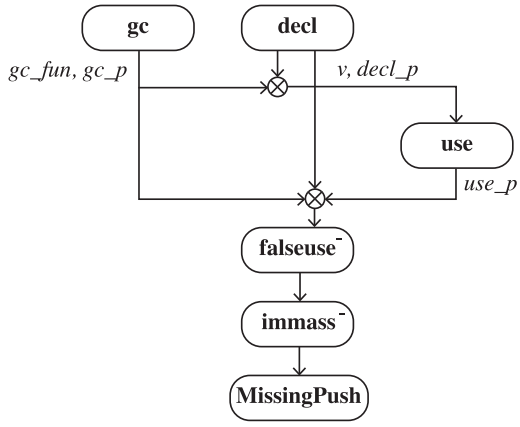

Fig. 8 Dependency of rules.

In contrast, while the program shown in **Fig. 7** satisfies all the conditions of **gc**, **decl**, and **use**, a GC_PUSH is not necessary. There are two reasons for this:

- variable x is assigned on line 3, which is between GC on line 2 and the use of x on line 4, and
- assignment to y on line 2 is performed after the function call, which is the same situation as variable p in Fig. 2.

To exclude these cases, we do not detect them as bugs if one of the following conditions is satisfied. In the rest of this paper, we add a minus sign superscript to the names of negative conditions.

- **assign**$^-$: v is assigned between *gc_p* and *use_p*.
- **immass**$^-$: v is assigned in the same statement as *gc_p* but after *gc_p*.

The program of eJSVM sometimes passes the address of a variable to a function to take an extra return value, as the address of ret is passed on line 10 in Fig. 2. This works as an assignment. However, because we cannot guarantee that the callee function assigns to the variable, we do not consider such "out parameters" as assignments.

4.2 Design of Rule for Missing GC_PUSH Bug

We construct the rule to find the missing GC_PUSH bugs by combining small pieces of rules, as shown in **Fig. 8**.

We develop rules, each of which checks a condition **gc**, **decl**, **use**, or **immass**$^-$ that we enumerated in Section 4.1. Then, we develop a rule **MissingPush** that depends on *decl_p* and *gc_p*, which are bound in the preceding rules. **MissingPush** detects the case where a GC_PUSH is missing between the position of the variable declaration, *decl_p*, and the position of the call of a function that may cause GC, *gc_p*. We implement the **assign**$^-$ condition as a path condition of the wildcard of the **use** rule.

Because the rule for the **use** condition locates any occurrence of v, it also detects assignments to v and occurrences of v as arguments of GC_PUSHes and GC_POPs. To ignore these false positives, we add the following rule.

- **falseuse**$^-$: Ignore assignment to v and occurrences as argu-

```
@gc@
expression gc_fun;
position gc_p;
type pre.Context;
Context ctx;
@@
gc_fun@gc_p(..., ctx,...)

@decl@
identifier v;
expression e;
position decl_p;
type T:script:python(){is_pointer(T)};
@@
(
 T v@decl_p;
|
 T v@decl_p = e;
)

@use depends on gc && decl@
identifier decl.v;
expression e, gc.gc_fun;
position gc.gc_p, use_p;
@@
gc_fun@gc_p
... when != v = e
    when exists
v@use_p

@falseuse depends on use@
identifier pre.push, pre.pop, decl.v;
expression e;
position use.use_p;
@@
(
 v@use_p = e
|
 push(&v@use_p)
|
 pop(&v@use_p)
)

@immass depends on use@
identifier decl.v;
expression gc.gc_fun;
position gc.gc_p;
@@
v = <+... gc_fun@gc_p ...+>
```

Fig. 9 Rules to find missing GC_PUSH.

ments of GC_PUSHes and GC_POPs.

4.3 Implementation of Rule for Missing GC_PUSH Bug

In this section, we explain how our rules work. We first explain the rules preceding the **MissingPush** rule shown in **Fig. 9**. Then, we show in **Fig. 10** the **MissingPush** rule again, which appeared in Fig. 5, and explain it.

The **gc** rule matches function calls that take context arguments. In the meta-variable declaration

```
@MissingPush depends on use && !falseuse && !immass@
identifier pre.push, decl.v;
expression e, gc.gc_fun;
position gc.gc_p, decl.decl_p;
type decl.T;
@@
(
 T v@decl_p;
|
 T v@decl_p = e;
)
... when != push(&v)
    when exists
gc_fun@gc_p
```

Fig. 10 Rule to find missing GC_PUSH (same as Fig. 5).

```
type pre.Context;
Context ctx;
```

we declare to use type `Context`, which is defined in the `pre` rule so that it matches type `Context*`. Then, we declare meta-variable `ctx` of that `Context` type. The **gc** rule matches function calls that take this `ctx` as their argument. We use the `expression` type, which matches expressions, for `gc_fun` rather than the `identifier` type, which matches identifiers, so that `gc_fun` can match function pointers as well.

The **decl** rule matches declarations of variables of the JSValue type. The following declaration of a meta-variable `T` describes that `T` should match only the types that make the Python program `is_pointer(T)` true, where `is_pointer(T)` is defined so that `T` is of the JSValue type.

```
type T:script:python(){is_pointer(T)};
```

The **decl** rule matches the declaration of variables when `T` matches their types. Note that the rule contains two patterns, one of which has an initializer and the other does not.

The **use** rule matches programs that use v after a function call gc_p, where v and gc_p are bound in the **gc** and **decl** rules. The wildcard "`...`" between gc_p and the use of v is modified with "`when != v = e`" so that the rule does not match if the program satisfies the **assign**$^-$ condition. Note that the wildcard "`...`" is quantified with "`when exists`" so that this rule can find all the variables that are used on any execution path, even if there are branches after the function call gc_p.

The **falseuse**$^-$ rule matches the occurrences of a variable v that the **use** rule detected if they are the assignments to v or arguments of GC_PUSH or GC_POP.

The **immass**$^-$ rule matches assignments to v performed in the same statement as the function call at gc_p, but after it. The left-hand side of the pattern

```
v = <+... gc_fun@gc_p ...+>
```

matches expressions that have `gc_fun@gc_p`.

The **MissingPush** rule shown in Fig. 10 combines the rules mentioned above by the following dependency description.

```
depends on use && !falseuse && !immass
```

Because the **use** depends on the **gc** and **decl** rules, they are not listed explicitly, but the **MissingPush** rule depends on them indirectly. The **MissingPush** rule matches programs that have an execution path that does not contain GC_PUSH between a declara-

tion of a JSValue type variable and a function call that may cause GC. To represent this condition, the wildcard "..." is decorated with "when != push(&v)" and exists.

5. Experiment and Case Study

We checked the source code of eJSVM using the rules we developed in Section 4. The code was checked after preprocessing to remove the macros used in eJSVM. GC_PUSHes and GC_POPs were left in as function calls.

5.1 Accuracy

First, we examined how many bugs were missed by Coccinelle with our rules. It is difficult to examine the number of misses because we would need to know all the positions of bugs, so instead, we assumed that the source code of eJSVM contains GC_PUSHes and GC_POPs correctly. We removed all GC_PUSHes and GC_POPs from the source code and then tested if Coccinelle could detect the positions where they had been.

The source code of eJSVM used 33 GC_PUSHes. Coccinelle detected 27 out of these as missing. The remaining six were redundant, and they were not bugs in reality. An example of the positions where GC_PUSHes were redundant is shown in **Fig. 11**. In this program, nexth and oh are GC_PUSHed on the first line, but it is not possible for GC to take place until the corresponding GC_POPs are performed.

As we have shown, all of the necessary GC_PUSHes and GC_POPs can be found. Thus, we conclude that Coccinelle can find bugs at high accuracy.

Note that bugs other than the missing GC_PUSH bug appear only when there is a GC_PUSH or a GC_POP. Thus, only missing GC_PUSH bugs were found in this examination.

5.2 Finding Bugs from eJSVM

Next, we applied our rules to the source code of eJSVM to find its bugs. **Table 1** lists the result and **Fig. 12** visualizes its summary. The solid lines of "manual" and "Coccinelle" in Fig. 12 represent the number of GC_PUSHes that the developer inserted and GC_PUSHes reported by Coccinelle when we removed GC_PUSHes, respectively. The columns "correctly inserted", "bugs", and "FP" in Table 1 show the number of GC_PUSHes that developers correctly inserted, the missing GC_PUSHes found using Coccinelle, and GC_PUSHes reported by Coccinelle but that were not necessary, respectively. The segment "redundant" in Fig. 12 represents these six redundant GC_PUSHes that were inserted by the developer. Note that we did not find any bugs other than the missing GC_PUSHes.

Coccinelle reported quite a few missing GC_PUSHes. We examined all reported positions and found that all of them, including the code shown in Fig. 4, were necessary, apart from a single false positive.

The false positive occurred in a case where the possible combinations of branches to be taken in the two if statements were limited by control variables. **Figure 13** shows the relevant piece of code, where lowerValue and upperValue are of the JSValue type. Coccinelle reported that GC_PUSHes for both of them were missing, but in fact, upperValue did not need to be GC_PUSHed.

```
1  GC_PUSH(nexth); GC_PUSH(oh);
2  r = hash_put_with_attribute(
3          hidden_map(nexth),
4          name, index, attr);
5  if (r != HASH_PUT_SUCCESS) {
6    GC_POP(oh); GC_POP(nexth);
7    return FAIL;
8  }
9  hidden_n_entries(nexth)++;
10 hash_put_with_attribute(
11     hidden_map(oh), name,
12     (HashData)nexth,
13     ATTR_NONE | ATTR_TRANSITION);
14 GC_POP(oh); GC_POP(nexth);
```

Fig. 11 Redundant GC_PUSH.

Table 1 Number of missing GC_PUSHes with lines of code (LoC), number of correctly inserted GC_PUSHes by hand, and number of false positives (FP).

file name	LoC	correctly inserted	bug	FP
allocate.c	228	3	0	0
builtin-array.c	797	3	23	1
builtin-boolean.c	73	0	3	0
builtin-global.c	277	0	2	0
builtin-math.c	257	0	1	0
builtin-number.c	175	0	3	0
builtin-object.c	110	0	2	0
builtin-regexp.c	273	0	0	0
builtin-string.c	692	6	6	0
call.c	251	0	0	0
codeloader.c	761	0	0	0
context.c	168	1	0	0
conversion.c	669	0	6	0
gc.c	1066	0	0	0
hash.c	474	0	0	0
init.c	114	0	0	0
main.c	429	0	0	0
object.c	1030	14	18	0
string.c	189	0	0	0
vmloop.c	1426	0	0	0
Total	9459	27	64	1

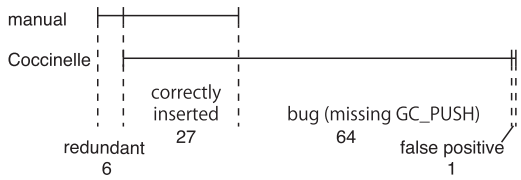

Fig. 12 Visualized summary of Table 1.

upperValue was reported because

- the call of set_array_prop on line 3, which may cause GC, satisfied the **gc** condition,
- the use of upperValue on line 9 satisfied the **use** condition, and
- between them, there existed an execution path that did not have any assignment to upperValue, which were taken if the condition of the if statement on line 5 did not hold.

However, it was only when upperExists was false that the condition of the if statement on line 5 did not hold. In this case, branches that used upperValue, such as line 9, was not taken in the following execution. Coccinelle performs a pattern match against control flow graphs, so control flows depending on data are handled conservatively. Hence, it causes false positives. However, this kind of code is likely to be hard to read for programmers, so we assume is is not written frequently.

```
1  for (lower = 0; lower < mid; lower++) {
2    if (lowerExists)
3      lowerValue = get_array_prop(context, args[0], cint_to_fixnum(lower));
4    upperExists = has_array_element(args[0], upper);
5    if (upperExists)
6      upperValue = get_array_prop(context, args[0], cint_to_fixnum(upper));
7
8    if (lowerExists && upperExists) {
9      set_array_prop(context, args[0], cint_to_fixnum(lower), upperValue);
10     set_array_prop(context, args[0], cint_to_fixnum(upper), lowerValue);
11   } else if (!lowerExists && upperExists) {
12     set_array_prop(context, args[0], cint_to_fixnum(lower), upperValue);
13     delete_array_element(args[0], upper);
14   } else if (lowerExists && !upperExists) {
15     set_array_prop(context, args[0], cint_to_fixnum(upper), lowerValue);
16     delete_array_element(args[0], lower);
17   } else {
18     /* No action is required */
19   }
20 }
```

Fig. 13 Program causing a false positive.

Table 2 Number of missing GC_PUSHes in AVL-tree program.

file name	LoC	correctly inserted	bug	FP
avltree.c	638	28	7	0

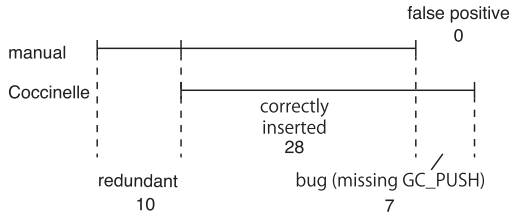

Fig. 14 Visualization of Table 2.

5.3 Checking Program Other Than eJSVM

We also tried to find missing GC_PUSH bugs in another program. We developed rules for the program in the same way as shown in Section 4. The program is a test program for a GC library developed by another research group and implements an AVL-tree in C whose nodes are allocated in the GC heap. It uses macros similar to the GC_PUSH of eJSVM to push the addresses of pointers to nodes on the GC root stack. In this program, any function can cause GC, excluding the debug functions.

Table 2 shows the result and **Fig. 14** visualizes its summary. The numbers in Table 2 do not include the ten redundant GC_PUSHes inserted by the developer. In this program, many GC_PUSHes were inserted conservatively. Nevertheless, Coccinelle found seven missing GC_PUSHes. We reported these bugs to the developer and confirmed that all of them were in fact missing GC_PUSHes.

6. Discussion

6.1 Limitations of Pattern Matching Approach against Control Flow Graph

Coccinelle performs pattern matching against control flow graphs. Thus, it may cause false positives if the execution paths are limited on the basis of the values of expressions. In our investigation, Coccinelle showed only a single false positive for eJSVM (Fig. 13), where combinations of the branches to be taken by two `if` statements were limited. In a program where whether or not a JSValue type variable is used depends on the value of a loop condition, a similar problem may occur.

In eJSVM, GC_POP receives the variable expected to be on the top of the GC root stack for debugging. In a debug build, the runtime system verifies that GC_POPs correspond to GC_PUSHes. This is difficult to check using Coccinelle. For example, if variables a and b are GC_PUSHed in this order and GC_POPed in the same order, it should be reported as a bug. However, we cannot develop a general rule to detect this kind of bug.

6.2 Automated Insertion of GC_PUSH

A rule for Coccinelle can rewrite the pieces of programs that a pattern matches. Thus, with this feature, a rule could insert GC_PUSHes after variable declarations automatically. However, this may produce a suboptimal program.

Inserting GC_PUSHes in optimal positions is difficult because they depend on how the program is used. If a GC_PUSH is inserted before the furthest node from the entry of the function among the nodes that dominate all the nodes that may cause GC, the resulting program would be close to the optimal. However, writing such complicated rules in SmPL increases the risk of bugs in the rule.

We could use a compiler infrastructure to find the missing GC_PUSH bugs and insert GC_PUSHes on its intermediate representation. However, this would require more effort than using an existing tool, such as Coccinelle.

6.3 Using Coccinelle in Development of eJSVM

As we have shown in Section 5.1, if we check a program from which all GC_PUSHes and GC_POPs are removed, Coccinelle can identify all the positions where GC_PUSHes are required. After this investigation, we removed all GC_PUSHes and GC_POPs from the source code of eJSVM and inserted only the necessary ones. In this task, the developers chose optimal positions to insert GC_PUSHes. This task was performed by a team consisting of an academic staff who is not an author of this paper and three students including one of the authors, and took about three hours.

We also added a production rule in Makefile of eJSVM so that our rules could be applied to check bugs whenever we modified

source code. Owing to this, we could find some bugs before they occurred. This check takes about one minute on a normal desktop computer, which is shorter than our regression test. Thus, the time to check is not an obstacle to our development.

7. Related Work

7.1 Management of Root Set

There are various techniques to find GC roots from local variables, including explicit pushing and popping, which eJSVM utilizes. Provided we can modify the compiler, it is a commonly used technique to generate stack maps when compiling the VM. A stack map is a data structure that tells the locations of pointers in function frames.

Henderson [6] proposed creating a structure for each function to hold all pointers to the heap. These structures are allocated on the stack and linked together so that GC can find all such structures by traversing the list. Henderson introduced these structures by program transformation. It is not realistic for programmers to introduce these structures manually because they have to be defined for each function.

A shadow stack is another well-known technique. A shadow stack is a separate stack from the control stack. Pointers to the heap are duplicated and placed on the shadow stack as well as on the control stack. The Java Native Interface (JNI) and the V8 JavaScript engine convert pointers to the heap into handles when they are stored in local variables. This technique has the risk of a bug that misses the release of handles. In C++, we can rely on destructors to release handles when the control exits a scope. In fact, V8 uses this technique.

Finally, conservative GC [7] is also a well-known technique. Conservative GC considers a pointer-like value, which has the same bit pattern as a pointer to an object, to be a pointer. Conservative GC, however, cannot move objects.

7.2 Finding Bugs in C Program

Coccinelle was developed for the Linux kernel project and is used for finding bugs and refactoring the Linux kernel [8], [9]. It is also used for open source software other than OS. For example, one study found bugs in OpenSSL [10]. The Coccinelle project provides many examples of rules, such as finding the NULL pointer dereference bug [*3]. However, our study is the first research to apply Coccinelle to GC, to the best of our knowledge.

Nishiwaki et al. [11] developed SEAN, a pattern matching tool against C programs to find bugs relating to misuse of JNI references. SEAN matches a certain pattern of an abstract syntax tree (AST). Nakamura et al. [12] generalized SEAN and developed ASTGrep, which searches for the pattern of AST that the user specified. Quinlan et al. [13] also developed a pattern matching tool to find bugs that is built on top of the Rose compiler infrastructure.

8. Summary

In this work, we found bugs related to the addition and removal of heap pointing pointers to the root set. We used Coccinelle, a pattern matching tool against control flow graphs. We examined the accuracy of this method by applying it to eJSVM source code from which we removed pieces of code to maintain the root set. The results showed that Coccinelle found all the removed pieces that were actually necessary. Furthermore, it found many bugs, which helped to debug the eJSVM. We also applied the same approach to a source code developed by another research project and were able to identify bugs. Only one false positive was found, in a program where execution paths are limited by the values of expressions, which caused a false positive because Coccinelle performs pattern matching against control flow graphs.

Acknowledgments The authors would like to thank all members involved in the eJS project. They would also like to thank the reviewer for valuable comments.

This work was supported by the JSPS KAKENHI Grant Number 16K00103.

[*3] http://coccinelle.lip6.fr/rules/

References

[1] Ugawa, T., Iwasaki, H. and Kataoka, T.: eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems, *Journal of Computer Languages*, Vol.51, pp.261–279 (2019).

[2] Kataoka, T., Ugawa, T. and Iwasaki, H.: A Framework for Constructing JavasSript Virtual Machines with Customized Datatype Representations, *Proc. SAC 2018*, pp.1238–1247, ACM (2018).

[3] Brunel, J., Doligez, D., Hansen, R.R., Lawall, J.L. and Muller, G.: A Foundation for Flow-based Program Matching: Using Temporal Logic and Model Checking, *Proc. 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL '09*), pp.114–126, ACM (2009).

[4] Padioleau, Y., Lawall, J., Hansen, R.R. and Muller, G.: Documenting and Automating Collateral Evolutions in Linux Device Drivers, *Proc. 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (*Eurosys '08*), pp.247–260, ACM (2008).

[5] Chambers, C., Ungar, D. and Lee, E.: An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes, *Proc. Object-oriented Programming Systems, Languages and Applications* (*OOPSLA '89*), pp.49–70, ACM (1989).

[6] Henderson, F.: Accurate Garbage Collection in an Uncooperative Environment, *Proc. 3rd international symposium on Memory management* (*ISMM '02*), pp.150–156, ACM (2002).

[7] Boehm, H.-J. and Weiser, M.: Garbage Collection in an Uncooperative Environment, *Software – Practice and Experience*, Vol.18, No.9, pp.807–820 (1988).

[8] Lawall, J.L., Muller, G. and Palix, N.: Enforcing the Use of API Functions in Linux Code, *Proc. 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pp.7–12, ACM (2009).

[9] Palix, N., Thomas, G., Saha, S., Calvès, C., Muller, G. and Lawall, J.: Faults in Linux 2.6, *ACM Trans. Computer Systems*, Vol.32, No.2, pp.4:1–4:40 (2014).

[10] Lawall, J., Laurie, B., Hansen, R.R., Palix, N. and Muller, G.: Finding Error Handling Bugs in OpenSSL Using Coccinelle, *2010 European Dependable Computing Conference*, pp.191–196 (2010).

[11] Nishiwaki, H., Ugawa, T., Umatani, S., Yasugi, M. and Yuasa, T.: SEAN: Support Tool for Detecting Rule Violations in JNI Coding, *IPSJ Trans. Programming*, Vol.5, No.3, pp.23–28 (2012).

[12] Nakamura, S., Ugawa, T. and Umatani, S.: A Code Checker That Uses Tree Patterns Reflecting the Structures of Rule Violation Code, *IPSJ Trans. Programming* (*PRO*), Vol.9, No.4, pp.1–15 (2016).

[13] Quinlan, D.J., Vuduc, R.W. and Misherghi, G.: Techniques for Specifying Bug Patterns, *Proc. 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, pp.27–35, ACM (2007).

Tomoharu Ugawa received his B.Eng. degree in 2000, M.Inf. degree in 2002, and Dr.Inf. degree in 2005, all from Kyoto University. He worked for a research project on real-time Java at Kyoto University from 2005 to 2008. In 2008–2014, he was an assistant professor at the University of Electro-Communications. He is currently an associate professor at Kochi University of Technology. His work is in the area of implementation of programming languages with a specific focus on memory management. He received the IPSJ Yamashita SIG Research Award in 2012.

Taiki Fujimoto was born in 1996. He received his B.E. degree from Kochi University of Technology in 2019. He received the Best Presentation Award from 2018 Shikoku-section Joint Convention of the Institutes of Electrical and related Engineers.