Analyzing Dynamic Code: A Sound Abstract Interpreter for *evil* eval

VINCENZO ARCERI*, University of Verona, Italy
ISABELLA MASTROENI, University of Verona, Italy

Dynamic languages, such as JavaScript, employ string-to-code primitives to turn dynamically generated text into executable code at run-time. These features make standard static analysis extremely hard if not impossible because its essential data structures, i.e., the control-flow graph and the system of recursive equations associated with the program to analyze, are themselves dynamically mutating objects. Nevertheless, assembling code at run-time by manipulating strings, such as by **`eval`** in JavaScript, has been always strongly discouraged since it is often recognized that *"eval is evil"*, leading static analyzers to not consider such statements or ignoring their effects. Unfortunately, the lack of formal approaches to analyze string-to-code statements pose a perfect habitat for malicious code, that is surely evil and do not respect good practice rules, allowing them to hide malicious intents as strings to be converted to code and making static analyses blind to the real malicious aim of the code. Hence, the need to handle string-to-code statements approximating what they can execute, and therefore allowing the analysis to continue (even in presence of dynamically generated program statements) with an acceptable degree of precision, should be clear. In order to reach this goal, we propose a static analysis allowing us to collect string values and to soundly over-approximate and analyze the code potentially executed by a string-to-code statement.

ACM Reference Format:
Vincenzo Arceri and Isabella Mastroeni. 2020. Analyzing Dynamic Code: A Sound Abstract Interpreter for *evil* eval. *ACM Trans. Priv. Sec.* 1, 1, Article 1 (January 2020), 35 pages. https://doi.org/10.1145/3426470

1 INTRODUCTION

The possibility of dynamically building code instructions as the result of text manipulation is a key aspect in dynamic programming languages. In this scenario, programs can turn text, which can be built at run-time, into executable code [53]. These features are often used in code protection and tamper resistant applications, employing camouflage for escaping attack or detection [46], in malware, in mobile code, in web servers, in code compression, and in code optimization, e.g., in Just-in-Time (JIT) compilers, employing optimized run-time code generation.

While the use of dynamic code generation may simplify considerably the *art and performance of programming*, this practice is also highly dangerous, making the code prone to unexpected behaviors and malicious exploits of its dynamic vulnerabilities, such as code/object-injection attacks for privilege escalation, database corruption, and malware propagation. It is clear that more advanced and secure functionalities based on string-to-code statements could be permitted if we better master how to safely generate, analyze, debug, and deploy programs that dynamically generate and manipulate code.

*This is the corresponding author

Authors' addresses: Vincenzo Arceri, University of Verona, 15, Strada Le Grazie, Verona, 37134, Italy, vincenzo.arceri@univr.it; Isabella Mastroeni, University of Verona, 15, Strada Le Grazie, Verona, 37134, Italy, isabella.mastroeni@univr.it.

2471-2566/2020/1-ART1 $15.00
https://doi.org/10.1145/3426470

```
vd, ac, la = "";
v = "wZsZ"; m = "AYcYtYiYvYeYXY";
tt = "AObyaSZjectB";
l = "WYSYcYrYiYpYtY.YSYhYeYlYlY";

while (i+=2 < v.length)
  vd = vd + v.charAt(i);

while (j+=2 < m.length)
  ac = ac + m.charAt(j);

ac += tt.substring(tt.indexOf("O"), 3);
ac += tt.substring(tt.indexOf("j"), 11);

while (k+=2 < l.length)
  la = la + l.charAt(k);

d = vd + "=new " + ac + "(" + la + ")";
eval(d);
```

Fig. 1. A potentially malicious obfuscated JavaScript program.

There are lots of good reasons to analyze programs building strings that can be later executed as code. An interesting example is code obfuscation. Recently, several techniques have been proposed for JavaScript code obfuscation[1], meaning that also client-side code protection is becoming an increasingly important problem to be tackled by the research community and by practitioners. Consider, for example, the JavaScript program fragment in Fig. 1 where strings are manipulated, de-obfuscated [27], combined together into the variable `d` and finally transformed into executable code, the statement `ws=new ActiveXObject(WScript.Shell)`. This command, in Internet Explorer, opens a shell which may execute malicious commands. The command is not hard-coded in the fragment but it is built at run-time and the initial values of `i`,`j` and `k` are statically unknown, such as the number of iterations of the loops in the fragment. Hence, it is not always possible to simply ignore **`eval`** without accepting to lose the possibility of analyzing the rest of the program.

The problem. A major problem in presence of dynamic code generation is that static analysis becomes extremely hard if not even impossible. This happens because program's essential data structures, such as the control-flow graph and the system of recursive equations associated with the program to analyze, are themselves dynamically mutating objects. In a sentence: *"You can't check code you don't see"* [10]. Indeed, the only *sound* way analyses have to overcome the execution of code they "don't see" is to suppose that a string-to-code statement can do *anything*, i.e., it can generate *any* possible memory. Hence, when reaching such a statement, an analysis may continue but by accepting to lose any previously gathered information. Let us show this situation on a simple but expressive enough example. Consider the code on the left, where there is a variable `x` independent from what is dynamically executed in `y`. Suppose we are interested in analyzing the range of the variable `x` inside the loop, i.e., at line 5. Executing the code, we can observe that the range of `x` at line 5 is precisely $[1, 9]$, and this would be the result of any interval analysis on the code without line 6. Unfortunately, the presence of **`eval`** makes it impossible, for existing analyses, to know whether there is any "statically hidden" (i.e., dynamically generated) modification of `x`, and therefore it cannot properly compute the interval of `x`. This is a very simple use of **`eval`**, nevertheless, it is not even suitable to code rewriting techniques removing (when

```
1  x = 1;
2  a = 1;
3  y = "a++;";
4  while (x<10)
5    y = concat(y, y);
6    eval(y);
7    x++;
```

[1]https://www.daftlogic.com/projects-online-javascript-obfuscator.htm,
http://www.danstools.com/javascript-obfuscate/,
http://javascript2img.com/,
https://javascriptobfuscator.herokuapp.com/,
https://javascriptobfuscator.com/

possible) **eval** by replacing it with equivalent code (without **eval**) [37, 47]. Indeed, in the example, the **eval** parameter is not *hard-coded* but dynamically generated.

It should be clear that, the only way to make the analysis aware of the fact that the execution of **eval** does not modify x is to compute, or at least to over-approximate, what is executed in **eval**. For this reason, first of all, we need an abstract domain for *collecting* the string values of variables, such as y in the example. Unfortunately, this is not sufficient, since once we have the language over-approximating the values of y we still have to "execute" this language for analyzing the potential effects on x. Hence, we also need to extract from any language the code (or an over-approximation of it) that could be executed when it is executed by **eval**. The idea, at this point, is that of (recursively) call the abstract interpreter, for the performed analysis, on this *approximated* code. In the example, we could over-approximate the language of y as an arbitrarily long concatenation of strings `"a++;"`[2]. Anyway, what is clear is that any code we can synthesize from this language cannot add any statement modifying x. In particular, the call of the analysis on the synthesized code will surely return a memory where a is changed, but such that the analysis of x can continue unaffected.

The **eval** *that criminals do.* It is well known that, in not malicious code, an improper use of **eval** may create vulnerabilities opening breaches for several kinds of attack. **eval** injection which may lead to DOM based XSS attacks, broken authentication and session management, security misconfiguration, sensitive data exposure are only examples of how an attacker can exploit an improper use of **eval**. For this reason, many researchers do not consider **eval** as a problem, since it is often recognized that *"eval is evil"*, justifying the standard approach for tackling dynamic code generation in programming, based on preventing or even banishing it. Hence, most existing analyses of dynamic languages do not consider string-to-code primitives [2], thus being inherently unsound for these languages, or implement ad-hoc or pattern-driven transformations in order to remove **eval** (*removable* **eval**, i.e., **eval** whose argument is statically known, is, so far, the most widespread use of **eval**) [37, 47]. But this means to fix restrictions on the expressiveness of development tools, and may make tools blind to possible attacks exploiting **eval**, that usually do not respect good practice rules. Hence, it should be clear that this provides the perfect habitat where malware itself can use **eval**, being, most of existing analyses, defeated by the presence of this statement in the code. In other words, by ignoring **eval** or by considering useless to analyze **eval** which is not *removable*, we leave to malicious agents the possibility to use a powerful tool for generating dynamic code that naturally obfuscates code by making it not analyzable. For instance, we have analyzed[3] the collection of JavaScript malware provided in [1]. This collection contains more than 40.000 samples of JavaScript malware divided in folders. We have analyzed the 2017 folder, being the most recent malware collection of the considered benchmark. By analyzing this folder (containing 192 samples of malware), we have observed that at least 53% of malware contains calls to **eval**. Moreover, at least 23% of them contains implicit **eval** calls, meaning that the call is not readable and mentioned in the code but obfuscated, for example by string obfuscation. Analyzing all the malware samples, our analysis detected 121 **eval** calls, and, in all of them, the argument is created by manipulating strings (loops, string operations, string constructor), making the **eval** not removable. The JavaScript program fragment in Fig. 1 is a (simplified) example of such a malware. Moreover, **eval** (dynamic code in general) can be used also for protecting code: there already exist obfuscator tools[4] that may transform a *removable* **eval** [37] in an **eval** that cannot be removed, providing also malware with powerful obfuscation techniques against existing JavaScript analyzers. For all these reasons, we believe that the analysis of programs that dynamically transform strings into executable statements is something that cannot be ignored anymore.

[2]It is an *over*-approximation since in the concrete execution we execute at most 9 concatenations of the string.
[3]We have analyzed dynamic traces executions and, for each malware sample, we have collected the number of performed **eval** calls, tracked if each call occur explicitly or not, and collected the depth size of nested calls, if present.
[4]https://www.daftlogic.com/projects-online-javascript-obfuscator.htm,
http://www.danstools.com/javascript-obfuscate/

Our idea. In this paper (that is an extended and revised version of [6]), we tackle the problem of analyzing dynamic code by *treating code as any other dynamic structure that can be statically analyzed by abstract interpretation* [19], and to treat the abstract interpreter as any other program function that can be recursively called.
In order to obtain this it is necessary to tackle three main issues:

String abstract domain design: Since we have to collect the strings that may be argument of an **`eval`**, we need to design an abstract domain for strings collecting, as much faithfully as possible, the set of possible values that a string variable may receive before **`eval`** executes it. It surely has to approximate the set of possible string values, hence it has to be a language, it has also to keep enough information for allowing us to extract code from it, but it has also to keep enough information for analyzing properties of string variables that are never executed by an **`eval`** during computation.

String executability analysis: Since we have to analyze the code potentially executed by an **`eval`**, we need to extract from the (abstract) argument of **`eval`** (i.e., from the collection of its potential arguments) an abstraction of the code that this collection may contain. It should be clear that this abstraction must be in a form that the analyzer can interpret.

Interpreter recursive call: Finally, we recursively call the abstract interpreter of the executable approximated code obtained by the previous phase.

In order to make it possible to recursively call the abstract interpreter, we have precisely to solve the first two issues. As far as the first issue is concerned, we choose regular (formal) languages as string abstraction, since they are both precise enough for analyzing string properties in general, and suitable (by considering their finite representation as finite state automata) for building algorithms able to extract/approximate the executable sub-language of the string abstraction when necessary, in presence of string-to-code statements.
As far as the second issue is concerned, we choose control-flow graph as code abstraction, where the abstraction relation is the semantic inclusion relation, i.e., a control-flow graph G_1 is more abstract than G_2 if the set of executions of G_1 contains the set of executions of G_2. In this way, we guarantee a sound abstraction of the code executed by **`eval`**. It is clear that we have to transform the automaton A, generated by the string analysis, in the control-flow graph over-approximating the executable strings recognized by the automaton A.
This provides us both, with a proof of concept that a sound approximation of the semantics of dynamically generated programs is possible in abstract interpretation, and with a static analyzer for a core dynamic language, containing non removable **`eval`** statements, that still have some limitations in terms of precision (as we will explain in Sect. 5.3) but which provides the necessary ground for studying more precise solutions to the problem. The choice of considering a core programming language is just for focusing the attention on the approach, namely on the analysis architecture and on the algorithms proposed. The proofs of our results are reported in Appendix A.

2 ANALYZING DYNAMIC LANGUAGES: THE INGREDIENTS

In this section, we focus on the problem of defining an abstract (collecting) semantics for dynamic programs, namely programs containing string-to-code statements such as **`eval`**. This means that, as observed in the introduction, we need a semantics able to collect strings, sufficiently precise for inferring an approximation of what could be executed by the string to code statement, but also not too complex, in order to guarantee the effectiveness of the analysis.

2.1 The language: μJS

In this paper, we propose a language-independent framework for analyzing dynamic code, considering an imperative core language plus **`eval`**, namely μJS in Fig. 2, in order to focus precisely on the language features that makes dynamic a language and not on other features.

$$
\begin{array}{rl}
\text{Exp} \ni \mathsf{e} ::= & \mathsf{a} \mid \mathsf{b} \mid \mathsf{s} \\
\text{AExp} \ni \mathsf{a} ::= & x \mid n \mid \texttt{lenght}(\mathsf{s}) \mid \texttt{num}(\mathsf{s}) \mid \mathsf{a} + \mathsf{a} \mid \mathsf{a} - \mathsf{a} \mid \mathsf{a} * \mathsf{a} \\
\text{BExp} \ni \mathsf{b} ::= & x \mid \texttt{true} \mid \texttt{false} \mid \mathsf{e} = \mathsf{e} \mid \mathsf{e} > \mathsf{e} \mid \mathsf{e} < \mathsf{e} \mid \mathsf{b} \wedge \mathsf{b} \mid \neg \mathsf{b} \\
\text{SExp} \ni \mathsf{s} ::= & x \mid \text{'}\sigma\text{'} \mid \texttt{concat}(\mathsf{s}, \mathsf{s}) \mid \texttt{substr}(\mathsf{s}, \mathsf{a}, \mathsf{a}) \\
\text{Comm} \ni \mathsf{c} ::= & {}^{\ell_1}\texttt{skip}^{\ell_2} \mid {}^{\ell_1}x := \mathsf{e}^{\ell_2} \mid {}^{\ell_1}\mathsf{c}; {}^{\ell_2}\mathsf{c}^{\ell_3} \mid {}^{\ell_1}\texttt{if}\ (\mathsf{b})\ \{{}^{\ell_2}\mathsf{c}^{\ell_3}\}\ \texttt{else}\ \{{}^{\ell_4}\mathsf{c}^{\ell_5}\}^{\ell_6} \mid {}^{\ell_1}\texttt{while}\ (\mathsf{b})\ \{{}^{\ell_2}\mathsf{c}^{\ell_3}\}^{\ell_4} \mid {}^{\ell_1}\texttt{eval}(\mathsf{s})^{\ell_2} \\
\mu\text{JS} \ni \mathsf{P} ::= & {}^{\ell_\iota}\mathsf{c}; {}^{\ell_2} \qquad \text{where } \mathsf{Id} \ni x \text{ (Identifiers)}, n \in \mathbb{Z}, \sigma \in \Sigma^*
\end{array}
$$

Fig. 2. Syntax of μJS

The language is quite standard, and each statement is annotated with a label $\ell \in Lab$ corresponding to the statement program point in P. Let ℓ_ι be a special label identifying the initial program point and ℓ_f be a special label identifying the final/exit program point. We refer to the labels of P with Lab_P.

2.2 Analyzing μJS programs

In this section, we recall the ingredients of the static analysis process and the necessary semantics notions for statements of μJS. The approach we use is quite standard, but we recall it here for fixing also the notation used in the rest of the paper.

In order to analyze a program $\mathsf{P} \in \mu\text{JS}$, we have to build a corresponding control flow graph [54] (CFG for short), which embeds the control structure in the graph and leaves in the blocks (or equivalently on the edges) only the manipulation of the states (assignments) and the guards. We follow [54] for the construction of the control flow graph, where each node is a program point, and each edge is labeled with a statement or a guard. Formally, given a program $\mathsf{P} \in \mu\text{JS}$, we define the corresponding CFG $\mathsf{G}_\mathsf{P} \triangleq \mathtt{CFG}(\mathsf{P}) = \langle Nodes_\mathsf{P}, Edges_\mathsf{P}, In_\mathsf{P}, Out_\mathsf{P} \rangle$ as the CFG whose nodes are the program points, i.e., $Nodes_\mathsf{P} \triangleq Lab_\mathsf{P}$, the input node (without incoming edges) is the entry program point, i.e., $In_\mathsf{P} \triangleq \ell_\iota$, the output node (without outgoing edges) is the last program point, i.e., $Out_\mathsf{P} \triangleq \ell_\mathsf{f}$, and the edges $Edges_\mathsf{P} \in Nodes_\mathsf{P} \times \mu\text{JS} \times Nodes_\mathsf{P}$ are inductively defined on P by the auxiliary function

$$
\begin{array}{rcl}
Edges({}^{\ell_1}\texttt{skip}^{\ell_2}) & = & \{\langle \ell_1, \texttt{true}, \ell_2 \rangle\} \\
Edges({}^{\ell_1}x := \mathsf{e}^{\ell_2}) & = & \{\langle \ell_1, x := \mathsf{e}, \ell_2 \rangle\} \\
Edges({}^{\ell_1}\texttt{if}(\mathsf{b})\{{}^{\ell_2}\mathsf{c}_1{}^{\ell_3}\}\texttt{else}\{{}^{\ell_4}\mathsf{c}_2{}^{\ell_5}\}^{\ell_6}) & = & \{\langle \ell_1, \mathsf{b}, \ell_2 \rangle,\ \langle \ell_1, \neg\mathsf{b}, \ell_4 \rangle,\ \langle \ell_3, \texttt{true}, \ell_6 \rangle,\ \langle \ell_5, \texttt{true}, \ell_6 \rangle\} \\
 & & \cup\ Edges({}^{\ell_2}\mathsf{c}_1{}^{\ell_3}) \ \cup\ Edges({}^{\ell_4}\mathsf{c}_2{}^{\ell_5}) \\
Edges({}^{\ell_1}\texttt{while}\ (\mathsf{b})\ \{{}^{\ell_2}\mathsf{c}^{\ell_3}\}^{\ell_4}) & = & \{\langle \ell_1, \mathsf{b}, \ell_2 \rangle,\ \langle \ell_1, \neg\mathsf{b}, \ell_4 \rangle,\ \langle \ell_3, \texttt{true}, \ell_1 \rangle\} \ \cup\ Edges({}^{\ell_2}\mathsf{c}^{\ell_3}) \\
Edges({}^{\ell_1}\mathsf{c}_1; {}^{\ell_2}\mathsf{c}_2{}^{\ell_3}) & = & Edges({}^{\ell_1}\mathsf{c}_1{}^{\ell_2}) \ \cup\ Edges({}^{\ell_2}\mathsf{c}_2{}^{\ell_3}) \\
Edges({}^{\ell_1}\texttt{eval}(\mathsf{s})^{\ell_2}) & = & \{\langle \ell_1, \texttt{eval}(\mathsf{s}), \ell_2 \rangle\}
\end{array}
$$

In Fig. 3 and Fig. 4 we report two examples of CFG generation, following the above rules. From this construction, it is clear that the language of the CFG edge labels is an intermediate language slightly different from the μJS grammar. Edge labels correspond to a primitive statement (i.e., an assignment or **eval**) or a boolean guard, namely they are generated by the following grammar:

$$\mu\text{JS}^{\text{CFG}} \ni \mathtt{l} ::= x := \mathsf{e} \mid \mathsf{b} \mid \texttt{eval}(\mathsf{s})$$

At this point, given a CFG G, we denote by $Nodes(\mathsf{G})$ its set of nodes, by $Edges(\mathsf{G}) \subseteq Nodes(\mathsf{G}) \times \mu\text{JS}^{\text{CFG}} \times Nodes(\mathsf{G})$ its set of edges, by $In(\mathsf{G})$ its (unique by construction) input node and by $Out(\mathsf{G})$ its (unique by construction) output node. Finally, let $Paths(\mathsf{G}) \triangleq \left\{ \mathtt{l}_0\mathtt{l}_1 \ldots \mathtt{l}_k \,\middle|\, \forall i \leq k. \langle \ell_i, \mathtt{l}_i, \ell_{i+1} \rangle \in Edges(\mathsf{G}), \ell_0 = In(\mathsf{G}), \ell_{k+1} = Out(\mathsf{G}) \right\}$ be the set of computations of the CFG G.

```
1if (B) {
  2x := "a=a+1;"3
} else {
  4x := "b=b+1;"5
};6
[Next]
```

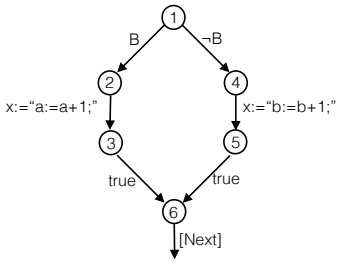


Fig. 3. Example of **if** CFG.

Our aim is to analyze μJS programs by analyzing their CFGs. Hence, first of all we have to specify the semantics associated with each possible edge of the CFG. In other words, we have to provide the semantics of the edge labels [54]. In particular, we have to formalize how each statement transforms a current state, which is represented as a store, namely as an association between identifiers and values. It is well known that, static program analysis works computing (abstract) collecting semantics, namely for each program point p and for each variable x, it computes the set of values that the variable x can have in any computation at the program point p. Hence, we define a (collecting) memory $\mathbb{m}$, associating with each variable a *set* of values instead of a single value. We define the set $\mathbb{M} \triangleq \mathsf{Var} \longrightarrow \wp(\mathbb{Z}) \cup \mathsf{Bool} \cup \wp(\Sigma^*)$, ranged over the meta-variable $\mathbb{m}$, where $\mathsf{Bool} = \wp(\{\textbf{false}, \textbf{true}\})$. We define two particular memories, $\mathbb{m}_\varnothing$ associating $\varnothing$ with any variable, and $\mathbb{m}_\top$ associating the set of all possible values with each variable. The update of memory $\mathbb{m}$ for a variable x with set of values v is denoted $\mathbb{m}[x/v]$. The partial order $\sqsubseteq$ between memories is defined as $\mathbb{m}_1 \sqsubseteq \mathbb{m}_2 \Leftrightarrow \forall x \in \mathsf{Id}.\ \mathbb{m}_1(x) \subseteq \mathbb{m}_2(x)$. Finally, lub and glb of memories are computed point-wise, i.e., $\mathbb{m}_1 \sqcup \mathbb{m}_2 \triangleq \lambda x.\ \mathbb{m}_1(x) \cup \mathbb{m}_2(x)$ and $\mathbb{m}_1 \sqcap \mathbb{m}_2 \triangleq \lambda x.\ \mathbb{m}_1(x) \cap \mathbb{m}_2(x)$.
The collecting (input/output) semantics of statements $\mathsf{c} \in \mu\mathrm{JS}^{\mathrm{CFG}}$ is defined as the function $[\![\mathsf{c}]\!] : \mathbb{M} \longrightarrow \mathbb{M}$. We denote by $(\!|\cdot|\!)$ the collecting semantics of expressions, defined as additive lift of the standard expression semantics.

$$
\begin{array}{rcl}
[\![x := \mathsf{e}]\!]\,\mathbb{m} & = & \mathbb{m}[x/(\!|\mathsf{e}|\!)\,\mathbb{m}] \\
[\![\mathsf{b}]\!]\,\mathbb{m} & = & \mathbb{m} \sqcap \bigsqcup \left\{ \mathbb{m} \;\middle|\; (\!|\mathsf{b}|\!)\,\mathbb{m} = \textbf{true} \right\} \\
[\![\textbf{eval}(\mathsf{s})]\!]\,\mathbb{m} & = & \bigsqcup_{\mathsf{c} \in \mathsf{C}} [\![\mathsf{c}]\!]\,\mathbb{m} \qquad \text{where} \qquad \mathsf{C} \triangleq (\!|\mathsf{s}|\!)\,\mathbb{m} \Cap \mu\mathrm{JS}
\end{array}
$$

where $\Cap$ is the intersection in the set of μJS programs. Formally let $\mathcal{S}$ be the function mapping any sequence $(\Sigma^*)^*$ on its string counterpart on Σ^* (and, abusing notation, also its additive lift to sets of sequences), and let $\mathit{tocode}(\sigma)$ for $\sigma \in \Sigma^*$ be the function interpreting a string of chars as code, if possible (and as **skip** otherwise), then for any $L \subseteq \Sigma^*$ we define $L \Cap \mu\mathrm{JS} \triangleq \left\{ \mathit{tocode}(\sigma) \;\middle|\; \sigma \in L \cap \left\{ \mathcal{S}(\delta) \;\middle|\; \delta \in \mu\mathrm{JS} \right\} \right\}$. The so far defined semantics

```
1x := "";
2while (B) {
  3x := concat(x, "s:=s+1;")4
};5
[Next]
```

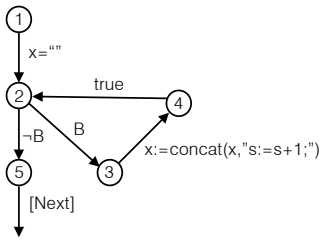


Fig. 4. Example of **while** CFG.

Algorithm 1 Static analysis on CFG of P.

Require: A CFG $\mathsf{G_P} = \langle Nodes_\mathsf{P}, Edges_\mathsf{P}, In_\mathsf{P}, Out_\mathsf{P}\rangle$
Require: A flow-sensitive input store $\mathbb{s}_0$
Ensure: $\mathbb{s}$ fix-point of the collecting memories for each program point (result of the analysis)

```
1: procedure ANALYZE(G_P, s_0)
2:     s ← s_0; s' ← ∅
3:     while s ≠ s' do
4:         s' ← s
5:         for ⟨ℓ_1, c, ℓ_2⟩ ∈ Edges_P do
6:             s ← s[s_ℓ2 / [[c]] s_ℓ1 ⊔ s_ℓ2]
7:         end for
8:     end while
9: end procedure
```

is standard for assignments and guards, while it says that when we meet an **eval** we have to extract, from the collection of strings for its argument s, only those strings corresponding to executable programs of μJS, we execute all these programs and we join all the resulting memories. It should be clear that, if the collecting semantics associated with the string expression s is an infinite set (it may happen in static analysis by approximation) then the (collecting) semantics of **eval** is undecidable. We can extend this definition of semantics to paths in a CFG $\mathsf{G_P}$: Let $\pi \in Paths(\mathsf{G_P})$, $\pi = \mathtt{l}_0\mathtt{l}_1 \ldots \mathtt{l}_k$, and $\mathbb{m} \in \mathbb{M}$ then $[\![\pi]\!]\,\mathbb{m} \triangleq [\![\mathtt{l}_k]\!] \circ \ldots \circ [\![\mathtt{l}_1]\!] \circ [\![\mathtt{l}_0]\!]\,\mathbb{m}$ [54]. Note that, given a program P, by construction of $\mathsf{G_P} = \mathtt{CFG}(\mathsf{P})$, it is well known [54] (and it can be easily proved by induction) that

$$\forall\, \mathbb{m} \in \mathbb{M}.\ \exists \Pi \subseteq Paths(\mathsf{G_P}).\ [\![\mathsf{P}]\!]\,\mathbb{m} = \bigsqcup_{\pi \in \Pi} [\![\pi]\!]\,\mathbb{m} \tag{1}$$

where the $[\![\mathsf{P}]\!]\,\mathbb{m}$ is the collection of the executions of P on the concrete memories collected in $\mathbb{m}$.

At this point, we use this semantics for analyzing μJS programs by computing the fix-point of the collecting semantics for each program point. In particular, we rewrite the standard fix-point algorithm for static analysis [51] in our notation. First of all, we define another important element, which is the collection of memories for each program point, that we will call *flow-sensitive* store $\mathbb{S} \triangleq Lab_\mathsf{P} \longrightarrow \mathbb{M}$ associating with each program point a (collecting) memory. Hence, a store $\mathbb{s} \in \mathbb{S}$ is a sequence of memories, one for each program point. We use $\mathbb{s}_\ell$ to denote $\mathbb{s}(\ell)$, namely the memory at program point ℓ. Given a store $\mathbb{s}$, the update of memory $\mathbb{s}_\ell$ with a new memory $\mathbb{m}$ is denoted $\mathbb{s}[\mathbb{s}_\ell/\mathbb{m}]$ and provides a new store $\mathbb{s}'$ such that $\mathbb{s}'_\ell = \mathbb{m}$ while $\forall \ell' \neq \ell$ we have $\mathbb{s}'_{\ell'} = \mathbb{s}_{\ell'}$. Finally, let $\mathbb{s}_\varnothing$ be the initial flow sensitive store where all the memories associate with all the variables the empty set, i.e., $\forall \ell \in Lab_\mathsf{P}.\ \mathbb{s}_\varnothing(\ell) = \mathbb{m}_\varnothing$. Then the static analysis algorithm is Alg. 1, whose result is a store $\mathbb{s}$ such that for each $\ell \in Lab_\mathsf{P}$, we have that $\mathbb{s}_\ell$ is the fix-point collecting memory for the program point ℓ. In general, when we call the analyzer on a program, the input store is $\mathbb{s}_0 = \mathbb{s}_\varnothing$.

3 ABSTRACTING DYNAMIC LANGUAGE ANALYSIS

It is well known that Alg. 1 may diverge on concrete memories. This means that we need abstraction for guaranteeing loop analysis convergence, as it is usual in static program analysis. Unfortunately, this is sufficient to avoid divergence (e.g., by using speed up techniques such as widening) when the code is static, but when code is dynamic other aspects of computation, not controllable by data abstraction, may cause divergence.

We have already observed that the collection of potential executable strings reaching an **eval** argument may be infinite, implying that, precisely as it happens for data values, we need to abstract also code (by suitable finite representations of potential infinite programs) in order to be able to enforce convergence by losing precision.

Finally, there is another potential subtle source of divergence due to the unpredictability of the code to execute in dynamic languages. Let us consider the code below.

ℓ_1`x := "eval(x)";` ℓ_2`eval(x);` ℓ_3

In this case, the second statement actives an infinite nested call chain to **eval**. This divergence comes directly from the meaning of dynamically generated code from strings and cannot be controlled by the semantics once we execute the string-to-code statement, since it is due to the generation of an infinite program and not to an infinite execution of a finite program.

In the following, we tackle these three problems separately, by suitably abstracting data, and in particular strings, relying on the already defined finite state automata abstract domain proposed in [5] and preparing the field for analyzing **eval** (Sect. 3.1); by abstracting/approximating code executed by **eval** in order to recursively call the analysis algorithm on the abstracted code and continue the analysis (Sect. 3.2 and 4); and by controlling the **eval** nested calls depth (Sect. 3.3).

3.1 Abstracting data

For solving the first standard source of divergence, we have to consider a suitable abstraction of data. In particular, we have to combine an abstraction of numerical values, of booleans and of strings. For the first two data types, the choice is not relevant in presence of string-to-code statements, except for tuning precision. In particular, a good choice (the one we made in the μJS analyzer) is the well known interval domain Int for numerical values [22] (equipped with widening for avoiding computation divergence), and the identity on boolean values Bool (which is a finite domain).

As far as strings are concerned, the issue is more complicated, since we need to collect string values during computation in order to be able to extract and approximate what is executable when a string-to-code statement, such as **eval**, is reached. Therefore, the resulting domain should have to approximate the set of possible string values, hence it has to be a language, and it has to keep enough information for allowing us to extract code from it, but it has also to keep enough information for analyzing properties of string variables that are never executed by an **eval** during computation.

We believe that a good choice, meeting all these requirements, are finite state automata (regular languages), since regular languages are precise enough for analyzing string properties in general, and since their finite representation (by means of finite state automata) is suitable for building algorithms able to extract/approximate the executable strings of the recognized language when necessary, namely in presence of string-to-code statements.

The finite state automata (FA) abstract domain has been introduced and implemented in [5] (and extended in [7]) for analyzing a generic imperative language manipulating strings. A FA A is a tuple $(Q^A, \delta^A, q_0^A, F^A, \Sigma^A)$, where Q^A is the set of states, $\delta \subseteq Q^A \times \Sigma^A \times Q^A$ is the transition relation, $q_0^A \in Q^A$ is the initial state, $F^A \subseteq Q^A$ is the set of final states and Σ^A is the finite alphabet of symbols. In order not to clutter the notation, when it is clear from the context, we refer to the FA A simply as $(Q, \delta, q_0, F, \Sigma)$. Given a state $q \in Q$, we denote by $\mathscr{L}_q(A)$ the language of the strings readable from the initial state q_0 to q. The language accepted by A is $\mathscr{L}(A) \triangleq \bigcup_{q \in F} \mathscr{L}_q(A)$. Given two FA A_1 and A_2 we have that $A_1 \equiv A_2$ iff $\mathscr{L}(A_1) = \mathscr{L}(A_2)$.

The finite state automata abstract domain is $\mathsf{FA}_{/\equiv}$, and its elements are the equivalence classes $[A]_\equiv$ of FA recognizing the same language, ordered w.r.t. language inclusion $\mathsf{FA}_{/\equiv} = \langle [A]_\equiv, \leq_{FA} \rangle$, where $[A_1]_\equiv \leq_{FA} [A_2]_\equiv$ iff $\mathscr{L}(A_1) \subseteq \mathscr{L}(A_2)$. The concretization in the domain of string properties, i.e., in the domain of languages, is the domain of regular languages. By Myhill-Nerode theorem [25] this domain is well defined and we can use the minimal automaton to represent each equivalence class. Moreover, the ordering relation is well defined since it does not depend on the choice of the FA used to represent the equivalence class. In particular, let Σ be the finite alphabet on which the automata in $\mathsf{FA}_{/\equiv}$ are defined, then for each $A \in \mathsf{FA}_{/\equiv}$, we have $\mathscr{L}(A) \in \wp(\Sigma^*)$. FA are

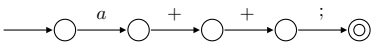


Fig. 5. Abstraction of x at line 2.

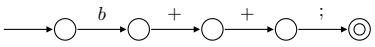


Fig. 6. Abstraction of x at line 4.

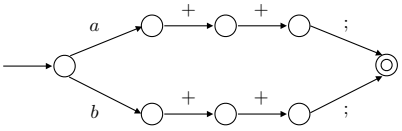


Fig. 7. Abstraction of x at line 5, at the **if** join point.

closed under finite language intersection and union, but unfortunately they do not form a Galois connection with $\wp(\Sigma^*)$. This means that it is not an abstract domain in the standard sense [19], but nevertheless, as stated in [20], this is not a concern since one can weaken the relation between concrete and abstract objects (and semantics) imposed by Galois connection, still ensuring soundness. In particular, in [5], the authors have proved that finite state automata domain can be used to analyze string manipulation programs. Indeed, starting from regular initial conditions, as it usually happens in static analysis, string analysis based on this domain will always compute regular invariants, since the domain is closed under common string manipulation operations [5, 7].
In order to provide the intuition of how this domain works for collecting strings we show an example where the union operation is used for getting a resulting collection of string values.

Example 3.1. Consider the μJS fragment in Fig. 3. In this case, we need to merge FA in the join point at the exit of the **if**-statement. The line 2 abstracts the value of x into the FA reported in Fig. 5. Similarly, the line 4 abstracts the value of x into the FA reported in Fig. 6. Since the boolean value of the **if**-guard is statically unknown, the analyzer must take into account, at the line 6, both the possible evaluations, performing the least upper bound (i.e., the automata union) between the two FA (Fig. 7).

Unfortunately, $\mathsf{FA}_{/\equiv}$ does not satisfy the ascending chain condition (ACC), hence fix-point computations of this domain may diverge. Therefore, in order to enforce convergence, we need to consider a widening[5] on $\mathsf{FA}_{/\equiv}$. The widening operator on $\mathsf{FA}_{/\equiv}$ is defined in terms of a widening operator over finite automata introduced in [28]. Let $A_1 = (Q^1, \delta^1, q_0^1, F^1, \Sigma^1)$ and $A_2 = (Q^2, \delta^2, q_0^2, F^2, \Sigma^2)$ be two FA such that $\mathscr{L}(A_1) \subseteq \mathscr{L}(A_2)$: the widening between A_1 and A_2 merges states that recognize the same language of length at most n, for some $n \in \mathbb{N}$. By changing the parameter n, we obtain different widening operators [28]. In particular, the parameter n tunes the length of the strings determining the equivalence of states, and therefore used for merging them in the widening, hence the smaller is n, the more information will be lost by widening automata. In the following, given two FA A_1 and A_2 with no constraints on the languages they recognize, we define the widening operator parametric on n on $\mathsf{FA}_{/\equiv}$ as $A_1 \nabla_n A_2$. As an example, we show what happens when computing FA in loops.

Example 3.2. Consider the μJS fragment in Fig. 4. The boolean value of the **while**-guard is statically unknown, as the number of loop iterations. After the first iteration the abstract value of x is the automaton A_1 corresponding to the sub-automaton of the one in Fig. 8 from state q_0 to state q_4. After the second iteration, the abstract value of x is the automaton A_2, namely the whole one depicted in Fig. 8. In order to enforce convergence we apply, in this example, the widening ∇_3. As we have already mentioned before, the widening ∇_3 merges together the states reading the same string of length 3. Hence, q_0 and q_4 (corresponding to the state p_0 in Fig. 9) and q_1 and q_5 (corresponding to the state p_1 in Fig. 9) are merged, while the other states remain singletons (q_2 and q_3) or

[5]A *widening operator* $\triangledown : A \times A \to A$ approximates the lub, i.e., $\forall x, y \in A. x, y \leq_A (x \triangledown y)$ and it is such that for any increasing chain $x_1 \leq x_2 \leq \cdots \leq x_n \leq \ldots$ the increasing chain $w^0 = \bot$ and $w^{i+1} = w^i \triangledown x_i$ is finite.

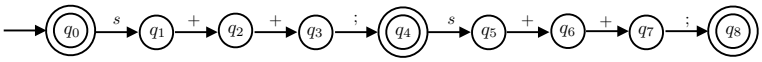


Fig. 8. Abstract value of x at line 2, after two loop iterations.

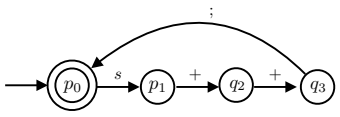


Fig. 9. Stable abstract value of x at line 2, after the widening application.

are removed by the minimization algorithm (q_6 and q_7). After the merge operation, the transitions are added to the resulting automaton: note that by adding the transition from q_3 to q_4, labeled with a semi-colon, a cycle is created in the new automaton, due to the merge operation of q_0 and q_4 in p_0. The resulting minimal automaton is shown in Fig. 9 and it encodes any possible repetition of the statement concatenations.

In the previous examples, we have provided the intuition of how the $\mathsf{FA}_{/\equiv}$ domain works for collecting string values during an analysis. The abstract semantics on $\mathsf{FA}_{/\equiv}$ for all the string operations and the corresponding soundness and completeness proofs can be found in [5, 7]. In particular, completeness is guaranteed by the `substr`, `charAt` and `concat` abstract semantics. The precision of the abstract semantics of the remaining operations (e.g., `indexOf`) can be further improved by applying the methodologies discussed in [8].

At this point, we need to combine all the three domains, Int, Bool and $\mathsf{FA}_{/\equiv}$. It is worth noting that, the way we combine these domains is not relevant for the dynamic feature of the language, but it may be relevant for other language aspects (e.g., type juggling and dynamic typing). In this paper, we combine these abstract domains by coalesced sum [4, 18]. We denote by $\mathsf{Int} \oplus \mathsf{Bool} \oplus \mathsf{FA}_{/\equiv}$ the coalesced sum abstract domain between intervals, booleans and automata, with the concretization function denoted by $\gamma_{\mathsf{v}} : \mathsf{Int} \oplus \mathsf{Bool} \oplus \mathsf{FA}_{/\equiv} \longrightarrow \wp(\mathbb{Z}) \cup \mathsf{Bool} \cup \wp(\Sigma^*)$ and defined as follows.

$$\gamma_{\mathsf{v}}(a) \triangleq \begin{cases} \varnothing & \text{if } a = \bot \\ \left\{ n \in \mathbb{N} \,\middle|\, i \leq n \leq j \right\} & \text{if } a \in \mathsf{Int} \wedge a = [i, j] \\ \mathscr{L}(a) & \text{if } a \in \mathsf{FA}_{/\equiv} \\ a & \text{if } a \in \mathsf{Bool} \\ \wp(\mathbb{Z}) \cup \mathsf{Bool} \cup \wp(\Sigma^*) & \text{otherwise} \end{cases}$$

Lub and glb on abstract values are denoted by $\sqcup^{\#}_{\mathsf{v}}$ and $\sqcap^{\#}_{\mathsf{v}}$, respectively. In the following, we denote by $\mathbb{m}^{\#} \in \mathbb{M}^{\#}$ the abstract memories, associating with variables values in the abstract domain just described i.e., $\mathbb{M}^{\#} \triangleq \mathsf{Var} \longrightarrow \mathsf{Int} \oplus \mathsf{Bool} \oplus \mathsf{FA}_{/\equiv}$. Lub and glb between abstract memories, respectively denoted by $\sqcap^{\#}$ and $\sqcup^{\#}$, are computed pointwise, namely $\mathbb{m}^{\#}_1 \sqcup^{\#} \mathbb{m}^{\#}_2 \triangleq \lambda x.\ \mathbb{m}^{\#}_1(x) \sqcup^{\#}_{\mathsf{v}} \mathbb{m}^{\#}_2(x)$ and $\mathbb{m}^{\#}_1 \sqcap^{\#} \mathbb{m}^{\#}_2 \triangleq \lambda x.\ \mathbb{m}^{\#}_1(x) \sqcap^{\#}_{\mathsf{v}} \mathbb{m}^{\#}_2(x)$. Moreover, we denote by $(\!|\cdot|\!)^{\#}\, \mathbb{m}^{\#}$ the abstract expression semantics and by $[\![\cdot]\!]^{\#}\, \mathbb{m}^{\#}$ the input-output collecting semantics (defined in the paragraphs below Thm 3.3) and by $\gamma : \mathbb{M}^{\#} \longrightarrow \mathbb{M}$ the abstract memory concretization function, i.e. $\gamma(\mathbb{m}^{\#}) \triangleq \lambda x.\ \gamma_{\mathsf{v}}(\mathbb{m}^{\#}(x))$.

Theorem 3.3. *Let $\mathbb{m} \in \mathbb{M}$ and $\mathbb{m}^{\#} \in \mathbb{M}^{\#}$ be the abstract memory abstracting $\mathbb{m}$. The abstract semantics of μJS (excluded* `eval`*) is sound, namely* $\forall \mathsf{P} \in \mu JS$

$$\exists \Pi \subseteq \mathit{Paths}(\mathsf{G_P}).\ [\![\mathsf{P}]\!]\, \mathbb{m} \sqsubseteq \gamma\Big(\bigsqcup^{\#}_{\pi \in \Pi} [\![\pi]\!]^{\#}\, \mathbb{m}^{\#} \Big)$$

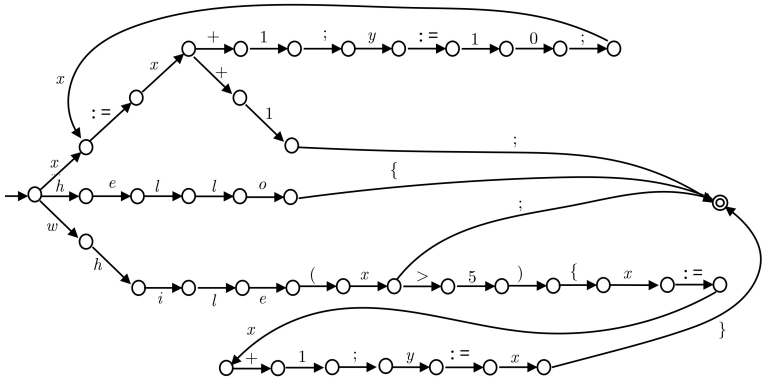


Fig. 10. FA A_{ds} abstracting the value of ds at program line 14 of Ex. 3.4.

Proof. The proof follows by soundness of the abstract expression semantics on intervals [19], automata and booleans expressions and the abstract semantics of statements [5]. □

Abstract semantics of **eval**. By considering the introduced abstract domain, as previously shown, we are able to compute a static analysis of strings, where the set of possible values of a string variable is collected and abstracted in a regular language, i.e., in a FA. In the concrete computation, **eval** turns strings into executable code, similarly, in the abstract one, we need to approximate the sub-language of only executable strings. The abstract semantics of **eval** becomes

$$[\![\textbf{eval}(\mathsf{s})]\!]^{\#}\,\mathbb{m}^{\#} = \bigsqcup^{\#}_{\mathsf{c}\in\mathbb{C}} [\![\mathsf{c}]\!]^{\#}\,\mathbb{m}^{\#} \qquad \text{where} \qquad \mathbb{C} \triangleq \mathscr{L}((\!|\mathsf{s}|\!)^{\#}\,\mathbb{m}^{\#}) \Cap \mu\mathsf{JS}$$

Let us provide the intuition by means of the following example.

Example 3.4. Consider the following μJS program P. For the sake of readability, we omit the **else** empty branches.

```
1while (x++ < 3)
  {2os := os + "xA:=Bx+1B;y:=1A0;x:=Bx+1A;"3};
4if (x > 10)
  {5os := "whiAleB(x>5A)A{x:A=x+1;y:=x};B"6};
7if (x = 5)
  {8os := "hello{"9};
10if (x = 8)
  {11os := "while(x;"12};
13ds := deobf(os);
14eval(ds);15
```

The statement `ds = deobf(os)` at line 13 is a syntactic sugar for the string transformer that removes the chars `"A"` and `"B"` from the string. In Fig. 10 we depict the abstract value of ds at the program point 14, computed analyzing strings on the $\mathsf{FA}_{/\equiv}$ domain, w.r.t. the widening ∇_3. The idea, at this point, is to remove from the automaton all the non-executable strings. This corresponds to perform the intersection between the (regular) language computed as the abstract value of ds (denoted by $\mathscr{L}((\!|\mathsf{ds}|\!)^{\#}\,\mathbb{m}^{\#})$ for a given memory $\mathbb{m}^{\#}$) and the (context-free) μJS language (also denoted by μJS):

$$\mathscr{L}((\!|\mathsf{ds}|\!)^{\#}\,\mathbb{m}^{\#}) \Cap \mu\mathsf{JS}$$

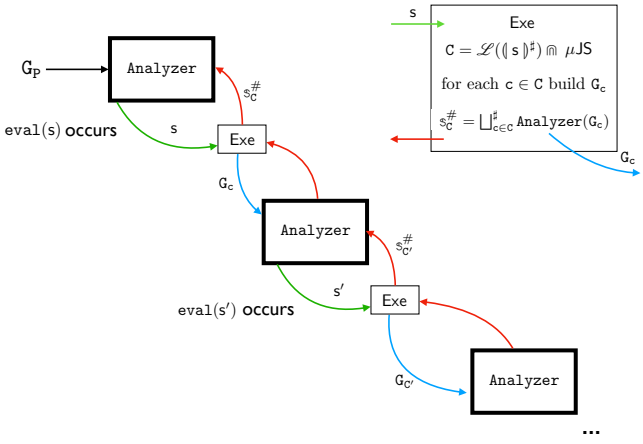


Fig. 11. Call execution structure of the analyzer

Unfortunately, it is well known that context-free languages are not closed under intersection, but nevertheless, the intersection between a context-free language and a regular language (which is our case) is always a context-free language. This means that we could indeed remove the non-executable automaton paths by performing an intersection such the one above, but unfortunately the computation of this intersection could be costly in practice due to the size of a real dynamic language grammar, such as for instance the JavaScript grammar.

Abstract semantics of assignments and boolean guards. We have just seen how we abstract the semantics of **eval**. The other labels in $\mu\mathrm{JS}^{\mathrm{CFG}}$ can be also abstracted on our abstract domain:

$$[\![x := \mathtt{e}]\!]^{\#}\, \mathbb{m}^{\#} = \mathbb{m}^{\#}[x/\langle\!|\mathtt{e}|\!\rangle^{\#}\, \mathbb{m}^{\#}] \qquad [\![\mathtt{b}]\!]\, \mathbb{m}^{\#} = \mathbb{m}^{\#} \sqcap^{\#} \bigsqcup^{\#} \left\{ \overline{\mathbb{m}^{\#}} \;\middle|\; \langle\!|\mathtt{b}|\!\rangle^{\#}\, \mathbb{m}^{\#} = \mathbf{true} \right\}$$

3.2 The analyzer structure

From the semantic point of view, it should be clear now how to handle the **eval** statement. We approximate the language of its argument, we extract from this a sound over-approximation of the sub-language of executable strings (by intersection), we execute from the current collection of memories all the obtained executable strings, we collect all the resulting collections of memories and from the union of these collection we continue the analysis.

The structure of the analyzer is shown in Fig. 11. In particular, when an **eval** is reached, the `Analyzer` relies on the Exe procedure, called on the current state. This procedure uses the abstract value (a FA) of the **eval** parameter for characterizing the code executed by **eval**. For instance, in Ex. 3.4, at program point 14, the procedure Exe computes the code to execute by using the automaton A_{ds} (Fig. 10). The way the procedure uses the FA, abstracting the **eval** parameter value, is defined by the above (naive) semantics of **eval**. When the analyzer has to compute $[\![\mathbf{eval}(\mathtt{s})]\!]^{\#}$, it calls the procedure Exe, which performs the intersection between the language grammar μJS and the regular language abstracting the value of the **eval** argument s, namely $\mathtt{C} = \mathscr{L}(\langle\!|s|\!\rangle^{\#}) \sqcap \mu\mathrm{JS}$. Afterwards, the Exe procedure recursively calls the `Analyzer` on the CFG $\mathtt{G_c}$ for each executable statement in C.
Finally, all the resulting stores from the analysis of each executable string of $\mathscr{L}(\langle\!|s|\!\rangle^{\#})$ are lubbed together, obtaining the collecting store $\mathbb{s}^{\#}_{\mathtt{C}} = \bigsqcup^{\#}_{\mathtt{c}\in\mathtt{C}} \mathtt{Analyzer}(\mathtt{G_c})$, that is returned to the analyzer that have called the procedure Exe. This process proceeds until we analyze a code without **eval**, whose analysis terminates.
As observed, this is a sound solution, but it may be undecidable since, due to widening, $\mathscr{L}(\langle\!|s|\!\rangle^{\#}) \sqcap \mu\mathrm{JS}$ may be an

infinite set of statements. Hence, we need a different approach allowing us to over-approximate this intersection by directly providing a finite representation of the code potentially executed by **eval**. For this reason, in Sect. 4, we construct an effective abstract procedure $\mathrm{Exe}^\#$ to call in place of Exe.

3.3 Abstracting sequences of nested calls to **eval**

In the previous section, we have described the structure of the analysis, which recursively calls the analysis on the code generated during computation. Due to unpredictability of the code that can be generated, it is impossible to foresee, from the program code, whether the recursive sequence of **eval** calls will terminate. At the beginning of this section, we have seen a quite simple example with a divergent recursion, but in general this kind of situations may be hard to detect and it is clearly out of the scope of the abstraction made on data (and of its widening). If the program using **eval** terminates, then there must be a maximal depth of nested calls to **eval**, and therefore we can ensure soundness until a maximal degree of nested calls to **eval**. However, to extract this maximal depth is in general undecidable.

In order to approximate this maximal depth of nested **eval** call, we can introduce a *nested call widening*, which consists of fixing a threshold of allowed depth of **eval** recursive calls. Once we have fixed this threshold (that can be decided by using statistical data, for instance in existing malware the maximal depth we observed was 3), we have two possibilities, depending on how confident we are on the good choice of the threshold. Once we reach the threshold, we could stop the recursive call sequence and simply continue the analysis from the current collection of memories, but in this case we can only be *soundy* [43], which means that we lose any guarantee of correctness about the values collected on the program points after the **eval** call.

Instead, the only way to keep soundness consists of approximating the collection of values for any program variable to the top, when the threshold is overcome, meaning that after the threshold anything can be computed. In this way, we guarantee soundness by fixing a degree of precision in observing the nesting of string-to-code statements.

It is clear that, in both cases we have a loss of precision in our analysis that has to balanced by the required computational efficiency. We will discuss these aspects in Sect. 5.3.

4 APPROXIMATING EXECUTABLE CODE

In the previous section, we observed that we have to characterize the sub-language of executable strings recognized by an automaton in an effective way. Moreover, as observed before, in concrete computations, **eval** turns strings into executable code, hence, once we have the sub-language of executable strings in the abstract domain, we need to turn finite state automata into executable code. Namely, we have to *synthesize* an approximation of a μJS program that is a sound approximation of the code that may be executed in the concrete execution. In particular, we provide an algorithmic approach for approximating in a decidable way the executability test $\mathscr{L}(\langle\!|\mathtt{s}|\!\rangle^\# \mathbb{m}^\#) \owedge \mu\mathrm{JS}$ (for a current abstract memory $\mathbb{m}^\#$), by building a CFG that soundly approximates the executable μJS programs in $\mathscr{L}(\langle\!|\mathtt{s}|\!\rangle^\# \mathbb{m}^\#)$, i.e., whose semantics soundly approximates the semantics of the code that may be executed into **eval**. This allows us to recursively call the abstract interpreter on the synthesized code.

In this section, we describe how we can provide a decidable procedure for extracting a sound approximation of executable code in the argument of an **eval** statement. This procedure will work by steps: Let $[\![\textbf{eval}(\mathtt{s})]\!]^\# \mathbb{m}^\#$ be the semantics the analyzer has to compute

(1) First, we have to clean up the language $\mathscr{L}(\langle\!|\mathtt{s}|\!\rangle^\# \mathbb{m}^\#)$ from all the strings that are surely not executable. This is obtained by visiting the automaton $A_\mathtt{s} = \langle\!|\mathtt{s}|\!\rangle^\# \mathbb{m}^\#$ and by keeping only those paths that can be executable. It should be clear that an automaton cannot recognize precisely a context free language, hence we still keep in the resulting automaton not executable strings, in particular those that do not respect the balanced bracketing. Let us denote the resulting automaton as $A_\mathtt{s}^{\mathtt{pStm}} \triangleq \mathtt{StmSyn}(A_\mathtt{s})$;

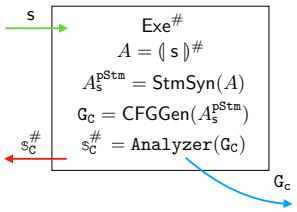


Fig. 12. Structure of Exe$^{\#}$, replacing Exe in Fig. 11.

(2) From the so far obtained automaton (A_s^{pStm}), the aim is now to build a CFG over-approximating the executable strings recognized by the automaton, i.e., $G_s \triangleq \mathsf{CFGGen}(A_s^{pStm})$. Then on this CFG the analyzer can be recursively called.

The new analyzer structure is obtained by replacing Exe with the new procedure Exe$^{\#}$ (reported in Fig. 12), in the structure in Fig. 11.

In order to understand an assumption made in the following construction of Exe$^{\#}$, we recall that the analysis process may add cycles in the FA due to the widening operation, and these cycles may involve different elements of the language. When a FA A_s is such that any cycle of A_s involves only executable strings of μJS we call it *cycle-executable*. For instance, the FA that accepts the language $\{\ (\texttt{x=x+1;})^n \mid\ n > 1\ \}$ is cycle-executable, while the one accepting $\{\ \texttt{x=(1)}^n\texttt{;} \mid\ n > 1\ \}$ is not. The reason why we need to make such a distinction is that, when a FA is not cycle-executable then we cannot extract, from the FA in the proposed framework, the CFG approximating the **eval** argument. Fortunately, this is a decidable condition on FA, and therefore, we first check whether the FA A_s, abstracting the **eval** argument, is cycle-executable, if it is not then we do not call Exe$^{\#}$ and we continue the analysis after the **eval** statement with the unknown value as result of **eval** (since, when we are not able to generate the CFG, we must suppose that any transformation may be executed by the **eval**). On the other hand, if the FA is cycle-executable then we call Exe$^{\#}$ for approximating the **eval** executed code. For this reason, in the following construction of Exe$^{\#}$, we make the assumption that the procedure is applied only to cycle-executable FA. Hence, both the approach and the implementation work for *any* FA, but they lose precision on not cycle-executable FA. In Sect. 5.3 we will discuss the problem and an idea for a potential improvement by means of an example.

4.1 StmSyn: Extracting of the executable code

The first step consists of reducing the number of states of the automaton, by over-approximating every string recognized as a statement, or partial statement, in μJS.

The idea is to derive, starting from the original automaton A_s (generated by the string analysis), whose alphabet is the set of characters Σ, a new automaton whose alphabet is a set of strings. These strings are obtained by collapsing consecutive edges, in A_s, up to any punctuation symbol in $\mathsf{Punct} \triangleq \{;, \{, \}, (,)\}$. In particular, any executable statement ends with a semicolon by language definition, the braces allow us to split strings when the body of a **while** or of an **if** either begins or ends, while the parentheses recognize the begin and the end of a parenthesized expression (the guard of an **if** or a **while**). In particular, we define a set of *partial statements*, that is a regular over-approximation of the μJS grammar, which will be the alphabet of the resulting automaton in a way such that the language on this alphabet contains μJS, namely it contains the regular super-language of μJS that can be recognized by a FA. The alphabet of *partial statements* is defined as follows

$$\Sigma_{\mathsf{pStm}} \triangleq \mathsf{Punct} \cup \left\{ x \in \Sigma^* \;\middle|\; \begin{array}{l} \text{x is a maximal substring of a } \mu\text{JS statement, between two} \\ \text{punctuaction symbols if present (only last punctuaction symbols included)} \end{array} \right\}$$

Lemma 4.1. *Let $\mathcal{S}$ be the function mapping any sequence over $(\Sigma^*)^*$ on its string counterpart on Σ^* (and, abusing notation, also its additive lift to sets of sequences), then $\mu JS \subseteq \mathcal{S}((\Sigma_{\texttt{pStm}})^*)$, namely any program $\texttt{P} \in \mu JS$ can be written as a sequence of partial statements in $\Sigma_{\texttt{pStm}}$. Formally,*

$$\forall \texttt{P} \in \mu JS\ .\exists k \in \mathbb{N}\ .\{\sigma_i\}_{i\in[0,k]} \in \Sigma_{\texttt{pStm}}\ .\texttt{P} = \mathcal{S}(\sigma_0\sigma_1 \dots \sigma_k).$$

At this point, the idea is that of transforming the automaton A_s on the alphabet Σ in the automaton $A_\text{s}^{\texttt{pStm}}$ on the alphabet $\Sigma_{\texttt{pStm}}$, losing in this way all the strings recognized by A_s which will be surely not executable. The soundness constraint obviously consists in guaranteeing that any executable string is not lost by this transformation. Unfortunately, $\Sigma_{\texttt{pStm}}$ is an infinite set, meaning that in general it cannot be used for defining a FA. Hence, we restrict this set to the set of partial statements which are substrings of strings recognized in the original automaton A_s on Σ. The idea is to perform a depth first visit on A_s in order to build the parsing tree $\texttt{T}_{A_\text{s}}$ of the automaton A_s, at this point we can define $\Sigma_{\texttt{pStm}}^{A_\text{s}} = \left\{\ x \in \Sigma_{\texttt{pStm}} \middle|\ x \text{ is a maximal path in } \texttt{T}_{A_\text{s}}\ \right\}$, where x is maximal if x is not a substring of any path in $\texttt{T}_{A_\text{s}}$ which is a partial statement in $\Sigma_{\texttt{pStm}}$.

Algorithm 2 Building the FA.

Require: A FA $A = (Q, \delta, q_0, F, \Sigma)$
Ensure: A FA $A' = (Q', \delta', q_0, F', \Sigma_{\texttt{pStm}}^{A_\text{s}})$
1: **procedure** StmSyn(A)
2: $Q' \leftarrow \{q_0\};\ F' \leftarrow F \cap \{q_0\};\ \delta' \leftarrow \emptyset$, Visited $\leftarrow \{q_0\}$;
3: stmsyntr(q_0);
4: **end procedure**
5: **procedure** stmsyntr(q)
6: $B \leftarrow \texttt{Build}(A, q)$;
7: Visited $\leftarrow$ Visited $\cup\{q\}$; $Q' \leftarrow Q' \cup \left\{\ p \middle|\ (\mathbf{a}, p) \in B\ \right\}$;
8: $F' \leftarrow Q' \cap F$; $\delta' \leftarrow \delta' \cup \left\{\ (q, \mathbf{a}, p) \middle|\ (\mathbf{a}, p) \in B\ \right\}$;
9: $W \leftarrow \left\{\ p \middle|\ (\mathbf{a}, p) \in B\ \right\} \setminus$Visited;
10: **while** $W \neq \emptyset$ **do**
11: select p in W ($W \leftarrow W \setminus \{p\}$);
12: stmsyntr(p);
13: **end while**
14: **end procedure**

In order to derive the automaton $A_\text{s}^{\texttt{pStm}}$, we design a procedure StmSyn, reported in Alg. 2, taking as input an automaton on Σ (i.e., A_s for **eval**(s)) and returning the automaton on a finite subset of $\Sigma_{\texttt{pStm}}$. The idea of Alg. 2 is to perform, starting from the initial state q_0 of A_s, a visit of the states recursively identified by Alg. 3 and to recursively replace the sequences of edges that recognize a symbol in $\Sigma_{\texttt{pStm}}^{A_\text{s}}$ with a single edge labeled by the corresponding string. Alg. 3 (namely $\texttt{Build}(A_\text{s}, q)$) scans, starting from the state q, the edges of the original automaton A_s and, when a punctuation symbol occurs or a final state is reached, it verifies whether the string read so far is in $\Sigma_{\texttt{pStm}}$, otherwise it is discarded: This *executability* check is performed at lines 13 and 16 (word.$\sigma \in \Sigma_{\texttt{pStm}}$) and ensures, for any state q of the automaton A_s, that I_q contains only (partial) statements of μJS. In particular, from the initial state q_0 of A_s we reach the states computed by $\texttt{Build}(A_\text{s}, q_0)$, and the corresponding read words (line 6). Then, we recursively apply stmsyntr, and in turn the procedure Build at line 6 of Alg. 2, to these states, following only those edges that we have not already visited.
For instance, in Fig. 13 we have the computation of $\texttt{StmSyn}(A_{\texttt{ds}})$, denoted by $A_\text{s}^{\texttt{pStm}}$. We can observe that the

Algorithm 3 Statements recognized from a state q.

Require: A FA $A = (Q, \delta, q_0, F, \Sigma)$
Ensure: I_q set of all pairs (partial statement,reached state)

```
1:  procedure Build(A, q)
2:      I_q ← ∅
3:      BUILDTR(q,ε,∅)
4:  end procedure
5:  procedure BUILDTR(q,word,Mark)
6:      Δ_q ← { (σ,p) | δ(q,σ) = p }
7:      while Δ_q ≠ ∅ do
8:          select (σ,p) in Δ_q (Δ_q ← Δ_q \ {(σ,p)})
9:          if (q,p) ∉ Mark then
10:             if σ ∉ Punct ∧ p ∉ F then
11:                 BUILDTR(p,word.σ,Mark∪{(q,p)})
12:             end if
13:             if σ ∈ Punct ∧ word.σ ∈ Σ_Syn then
14:                 I_q ← I_q ∪ {(word.σ,p)}
15:             end if
16:             if p ∈ F ∧ word.σ ∈ Σ_Syn then
17:                 I_q ← I_q ∪ {(word.σ,p)}
18:             end if
19:         end if
20:     end while
21: end procedure
```

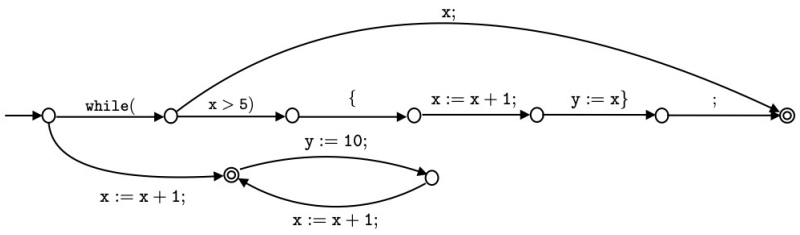
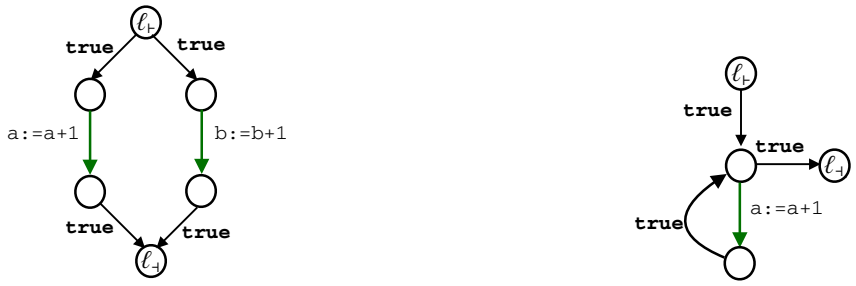


Fig. 13. Executable automaton $A^{\mathtt{pStm}}_{\mathsf{ds}} = \mathtt{StmSyn}(A_{\mathsf{ds}})$.

non-executable string `hello{` is not in $A^{\mathtt{pStm}}_{\mathsf{ds}}$ since it is discarded by Alg. 2, because it does not belong to $\Sigma_{\mathtt{pStm}}$. Instead, the string **`while`**`(x;` is still recognized by the resulting automaton even if it is not executable (this is due to the fact that with a FA we cannot recognize the balanced parenthesisation).

Lemma 4.2. *Let A be a cycle-executable finite state automaton and $A^{\mathtt{pStm}} = \mathtt{StmSyn}(A)$. If a string recognized by A corresponds to a partial statement, then the partial statement is recognized by $A^{\mathtt{pStm}}$. Formally, $\forall \sigma \in \Sigma^*_{\mathtt{pStm}}. \mathcal{S}(\sigma) \in \mathscr{L}(A) \Rightarrow \sigma \in \mathscr{L}(A^{\mathtt{pStm}})$.*

Next theorem tells us that any executable string collected during computation is kept in the transformed automaton, guaranteeing soundness.



(a) (b)

Fig. 14. Examples of CFG generation with the statically unknown guard ⊛.

Theorem 4.3. *Let* s *be a string expression, let* A_s *be the automaton recognizing the language of strings potentially associated with* s*, and* $A_s^{\texttt{pStm}} \triangleq \texttt{StmSyn}(A_s)$*, then* $\forall \sigma \in \mathscr{L}(A_s) \Cap \mu JS\,.\, \exists \delta \in \mathscr{L}(A_s^{\texttt{pStm}})$ *such that* $tocode(\mathcal{S}(\delta)) = \sigma$.

Proof. Given $\sigma \in \mathscr{L}(A_s) \Cap \mu\text{JS}$, from Lemma 4.1, $\exists \delta \in \Sigma^*_{\texttt{pStm}}$. $tocode(\mathcal{S}(\delta)) = \sigma$ and from Lemma 4.2, $\delta \in \mathscr{L}(A_s^{\texttt{pStm}})$. □

From the computational point of view, we can observe that the procedure $\texttt{Build}(A, q)$ executes a number of recursive-call sequences equal to the number of maximal acyclic paths starting from q on A. The number of these paths can be computed as $\sum_{q \in Q}(outDegree(q) - 1) + 1$, where $outDegree(q)$ is the number of outgoing edges from q. The worst case depth of a recursive-call sequence is $|Q|$. Thus, the worst case complexity of Build (when $outDegree(q) = |Q| \times |\Sigma|$ for all $q \in Q$) is $O(|Q|^3)$. As far as the procedure StmSyn is concerned, we can observe that in the worst case we keep in $\texttt{StmSyn}(A)$ all the $|Q|$ states of A, hence in this case we launch $|Q|$ times the procedure Build, and therefore the worst case complexity of StmSyn is $O(|Q|^4)$.

4.2 CFGGen: Control-flow graph generation

At this point, the idea is to use the so far obtained automaton over the alphabet $\Sigma_{\texttt{pStm}}$ to generate a CFG approximating the programs executed by **eval**(s). This phase is handled by the procedure CFGGen and works by several steps. It is well known that, an automaton can be equivalently rewritten as a regular expression r describing the language the automaton A recognizes [13], namely $\mathscr{L}(A) = \mathscr{L}(\text{r})$, where $\mathscr{L}(\text{r})$ denotes the language recognized by r. Let RE be the domain of regular expressions over the alphabet $\Sigma_{\texttt{pStm}}$, and $\texttt{Regex} : \text{FA} \longrightarrow RE$ be such an extractor. For instance, in the running example, $\text{r}_{\texttt{ds}} = \texttt{Regex}(A_{\texttt{ds}}^{\texttt{pStm}})$ is the following regular expression:

$$\text{r}_{\texttt{ds}} = \texttt{x:=x+1;} \parallel \textbf{while}\texttt{(x>5)\{x:=x+1;y:=x\};} \parallel \texttt{x:=x+1;(y:=10;x:=x+1;)}^* \parallel \textbf{while}\texttt{(x;}$$

where $\parallel$ and * respectively correspond to the disjunction and the Kleene-star between regular expressions. The analyzer implements the Brzozowski algebraic method [13] to convert an automaton to an equivalent regular expression. In particular, since concatenation is distributive w.r.t. $\parallel$, the conversion algorithm always distributes, in this case. Hence for instance, `x=1;(y =2;` $\parallel$ `y=3;)` is converted to `(x=1;y=2;)` $\parallel$ `(x=1;y=3;)`.

At this point, we pass through an augmented version of μJS before generating a CFG . We add, to the μJS boolean expressions, a statically unknown guard ⊛, namely b ::= . . . |⊛, that is intended as a boolean expression that both evaluates to **true** and **false** (i.e., it is statically unknown). We denote by $\mu\text{JS}^{\circledast}$ the μJS language plus ⊛. Let us show how we intend to use ⊛ in the CFG generation by means of the examples shown in Fig. 7a and Fig. 7b, corresponding to the CFG of **if**(⊛)`{a:=a+1}`**else**`{b:=b+1};` and **while**(⊛)`{a:=a+1};`, respectively. When ⊛ occurs

in a program, the CFG generator labels both the edges exiting from its program point with **true**. Let us focus on the **if** case. The static analysis algorithm on CFG (namely Alg. 1) must take into account both **if**-branches, emulating an abstract execution where the boolean guard is statically unknown. Similarly, in the **while** case, both the **true** and **false** branches of the **while** loop are labeled with **true**. In this way, we emulate a **while** loop where the boolean guard is statically unknown, i.e., the body must be executed an unbounded number of times.
Hence, we augment the function *Edges* in order to make the CFG generator (namely the function CFG) aware of the statically unknown guard ⊛ previously introduced.

$$\begin{aligned} \mathit{Edges}({}^{\ell_1}\texttt{if}(\circledast)\{{}^{\ell_2}\mathtt{c}_1{}^{\ell_3}\}\texttt{else}\{{}^{\ell_4}\mathtt{c}_2{}^{\ell_5}\}{}^{\ell_6}) &= \{\langle \ell_1, \texttt{true}, \ell_2\rangle,\ \langle \ell_1, \texttt{true}, \ell_4\rangle, \} \cup \{\langle \ell_3, \texttt{true}, \ell_6\rangle,\ \langle \ell_5, \texttt{true}, \ell_6\rangle\} \\ &\quad \cup\ \mathit{Edges}({}^{\ell_2}\mathtt{c}_1{}^{\ell_3})\ \cup\ \mathit{Edges}({}^{\ell_4}\mathtt{c}_2{}^{\ell_5}) \\ \mathit{Edges}({}^{\ell_1}\texttt{while}(\circledast){}^{\ell_2}\mathtt{c}{}^{\ell_3\,\ell_4}) &= \{\langle \ell_1, \texttt{true}, \ell_2\rangle,\ \langle \ell_1, \texttt{true}, \ell_4\rangle\} \cup \{\langle \ell_3, \texttt{true}, \ell_1\rangle\} \cup \mathit{Edges}({}^{\ell_2}\mathtt{c}{}^{\ell_3}) \end{aligned}$$

We abuse notation denoting by CFG the control-flow graph generator for $\mu\mathsf{JS}^{\circledast}$ programs, implementing the novel *Edges* rules reported above.

At this point, we have all the ingredients to generate a $\mu\mathsf{JS}^{\circledast}$ program from a regular expression over $\Sigma_{\mathtt{pStm}}$. In particular, we inductively define on the structure of regular expressions the function $\wr\cdot\int : RE \to \mu\mathsf{JS}^{\circledast}$ that, given $\mathsf{r} \in RE$, translates r to a $\mu\mathsf{JS}^{\circledast}$ program.[6]

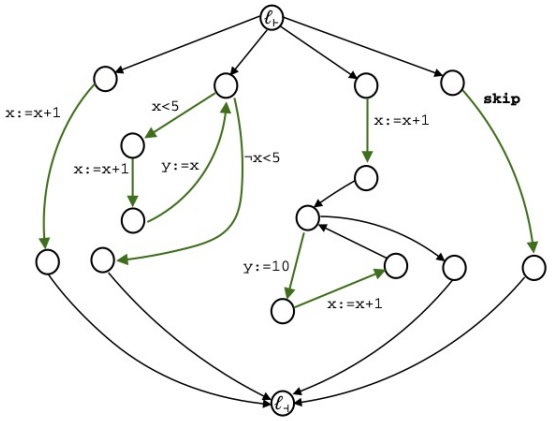
$$\begin{aligned} \wr\mathsf{d}\int &= \begin{cases} \mathit{tocode}(\mathcal{S}(\mathsf{d})) & \text{if } \mathsf{d} \in \Sigma_{\mathtt{pStm}} \\ \texttt{skip} & \text{otherwise} \end{cases} \\ \wr\mathsf{r}_1\mathsf{r}_2\int &= \wr\mathsf{r}_1\int\ \wr\mathsf{r}_2\int \\ \wr\mathsf{r}_1\|\mathsf{r}_2\int &= \texttt{if}(\circledast)\{\wr\mathsf{r}_1\int \in \mu\mathsf{JS}^{\circledast}\ \texttt{?}\ \wr\ \mathsf{r}_1\ \int\ \texttt{:skip}\}\texttt{else}\{\wr\mathsf{r}_2\int \in \mu\mathsf{JS}^{\circledast}\ \texttt{?}\ \wr\ \mathsf{r}_2\ \int\ \texttt{:skip}\} \\ \wr(\mathsf{r})^*\int &= \texttt{while}\ (\circledast)\ \{\wr\mathsf{r}\int \in \mu\mathsf{JS}^{\circledast}\ \texttt{?}\ \wr\ \mathsf{r}\ \int\ \texttt{:skip}\} \end{aligned}$$

In the base case (first line), we check whether d is a partial statement, namely if $\mathsf{d} \in \Sigma_{\mathtt{pStm}}$. If so, it is returned as code (abusing notation of *tocode*), otherwise **skip** statement is returned. In the case of $\wr\mathsf{r}_1\mathsf{r}_2\int$, the function concatenates the two programs inductively generated. In the case of $\wr\mathsf{r}_1\|\mathsf{r}_2\int$, we need to emulate the non-deterministic execution of both the operands. Here comes to play ⊛, previously introduced. In particular, we return as code an **if** statement, s.t. its boolean guard is ⊛, and where the **if**-true body is replaced with $\wr\mathsf{r}_1\int$ if it is executable, **skip** otherwise, and the **if**-false body is replaced with $\wr\mathsf{r}_2\int$, if it is executable, **skip** otherwise. Note that we need to check the executabilty of $\wr\mathsf{r}_1\int$ and $\wr\mathsf{r}_2\int$ since the function $\wr\cdot\int$ may return partial statements (hence, not executable statements) and the **if** statement, we aim to generate, must be executable, and in turn also the **true** and **false** bodies must be executable[7]. We treat in a similar way the case of $\wr(\mathsf{r})^*\int$: in order to guarantee soundness, the $\mu\mathsf{JS}^{\circledast}$ program $\wr\mathsf{r}\int$ must be executed an undefined number of times, hence, we build a **while** loop program, where the guard is ⊛. Finally, we define $\wr\mathsf{r}\int^P \triangleq \wr\mathsf{r}\int \in \mu\mathsf{JS}\ \texttt{?}\ \wr\ \mathsf{r}\ \int\ \texttt{:skip}$ that takes a regular expression r over partial statements and uses the auxiliary function $\wr\cdot\int$ previously defined to generate a $\mu\mathsf{JS}^{\circledast}$ program. In the following, we abuse notation denoting $\wr\cdot\int^P$ as $\wr\cdot\int$. In our running example, the code synthesis from the regular expression r_{ds} is the $\mu\mathsf{JS}^{\circledast}$ program reported in Fig. 15a.
The last step consists of generating a CFG on which we can recursively call our abstract interpreter. Hence, we call the function CFG on the synthesized code, namely $\mathtt{CFG}(\wr\mathsf{r}\int)$. In our running example, the CFG corresponding to the program reported in Fig. 15a is $\mathsf{G}_{\mathsf{ds}} = \mathtt{CFG}(\wr\mathsf{r}_{\mathsf{ds}}\int)$ reported in Fig. 15b, where the labels of consecutive **true** edges are omitted. Note that the CFG of **while**(x; corresponds to the CFG of **skip** (right-most path in Fig. 15b).

[6]We denote by b ? tt : ff the inline conditional construct, namely if b is true do tt, ff otherwise.
[7]The predicate $\wr\mathsf{r}\int \in \mu\mathsf{JS}^{\circledast}$ is decidable and it is checked at the call end to $\wr\mathsf{r}\int$.

```
if (⊛) {
   if (⊛) {
      if (⊛) {
         x := x + 1
      } else {
         skip
      }
   } else {
      while (x > 5) {
         x := x + 1; y := x
      }
   }
} else {
   x := x + 1;
   while (⊛) {
      y := 10; x := x + 1
   }
}
```

(a)



(b)

Fig. 15. (a) μJS$^{\circledast}$ program of $\wr \mathsf{r}_{\mathsf{ds}} \int$, (b) CFG G_{ds} generated by CFGGen module

Putting all the sub-procedures together, we can define the procedure CFGGen, that takes as input an automaton A over Σ_{pStm} and generates a CFG, as $\mathsf{CFGGen}(A) \triangleq \mathsf{CFG}(\wr \mathsf{Regex}(A) \int)$.

Finally, we need to prove soundness, namely we have to prove that the output CFG contains the computation of all the executable strings recognized by the starting automaton. Namely, we can show that the CFG generation does not lose any executable string. In particular, next lemma shows that the CFG generated by $\mathsf{CFG}(\wr \mathsf{r} \int)$ contains all the concrete computations of executable strings recognized by r, recalling that $\mathcal{S}$ converts a string of strings (in $(\Sigma^*)^*$) in a string of characters (in Σ^*), and *tocode* interprets a string of chars as a executable code, if possible.

Lemma 4.4. *Given a regular expression* $\mathsf{r} \in RE$ *over* Σ_{pStm}, *let* $\mathsf{G}_{\mathsf{r}} \triangleq \mathsf{CFG}(\wr \mathsf{r} \int)$, *then* $\forall \delta \in \mathscr{L}(\mathsf{r})$,

$$\forall\, \mathbb{m} \in \mathbb{M}.\ \exists \Pi \subseteq \mathit{Paths}(\mathsf{G}_{\mathsf{r}})\ \mathit{s.t.}\ [\![\mathit{tocode}(\mathcal{S}(\delta))]\!]\, \mathbb{m} \sqsubseteq \bigsqcup_{\pi \in \Pi} [\![\pi]\!]\, \mathbb{m}$$

Finally, next theorem tells us that any executable string collected by the analysis is kept in the final CFG.

Theorem 4.5. *Let* $\mathbb{m} \in \mathbb{M}$ *and* $\mathbb{m}^{\#} \in \mathbb{M}^{\#}$ *be the corresponding abstract memory. Let* $\mathsf{s} \in$ SExp *be a string expression and* A_{s} *be the FA recognizing the strings associated with* s *in the memory* $\mathbb{m}^{\#}$, *then* $\forall \sigma \in \mathscr{L}(A_{\mathsf{s}}) \Cap \mu JS.\ \exists \Pi \subseteq \mathit{Paths}(\mathsf{G}_{\mathsf{s}})$, $\mathsf{G}_{\mathsf{s}} \triangleq \mathsf{CFGGen}(\mathsf{StmSyn}(A_{\mathsf{s}}))$. $[\![\sigma]\!]\, \mathbb{m} \sqsubseteq \gamma\big(\bigsqcup^{\#}_{\pi \in \Pi} [\![\pi]\!]^{\#}\, \mathbb{m}^{\#}\big)$.

Proof. By Thm. 4.3, we have that $\forall \sigma \in \mathscr{L}(A_{\mathsf{s}}) \Cap \mu\mathsf{JS}$ there exists $\delta \in \mathscr{L}(\mathsf{StmSyn}(A_{\mathsf{s}}))$ such that $\mathit{tocode}(\mathcal{S}(\delta)) = \sigma$, hence any string collected in A_{s} corresponding to executable code, is kept in the transformed automaton $A_{\mathsf{s}}^{\mathsf{pStm}} = \mathsf{StmSyn}(A_{\mathsf{s}})$. By [13] it is well known that $\mathscr{L}(\mathsf{StmSyn}(A_{\mathsf{s}})) = \mathscr{L}(\mathsf{Regex}(\mathsf{StmSyn}(A_{\mathsf{s}})))$, hence $\delta \in \mathscr{L}(\mathsf{Regex}(\mathsf{StmSyn}(A_{\mathsf{s}})))$. By Lemma 4.4 we have $[\![\sigma]\!]\, \mathbb{m} = [\![\mathit{tocode}(\mathcal{S}(\delta))]\!]\, \mathbb{m} \sqsubseteq \bigsqcup_{\pi \in \Pi} [\![\pi]\!]\, \mathbb{m}$ and by Lemma 4.4 and Thm. 3.3 we have $\bigsqcup_{\pi \in \Pi} [\![\pi]\!]\, \mathbb{m} \sqsubseteq \gamma\big(\bigsqcup^{\#}_{\pi \in \Pi} [\![\pi]\!]^{\#}\, \mathbb{m}^{\#}\big)$. □

5 EVALUATING THE ANALYZER

We have implemented the μJS static analyzer (available at https://github.com/SPY-Lab/mujs-analyzer) described in this paper[8]. It should be clear that, being μJS a core language and not real JavaScript, our evaluation cannot

[8]It is worth noting that, we are currently integrating our approach in TAJS static analyzer [38].

be given in quantitative and efficiency terms, but it can only be given in terms of what it can analyze and how precisely. However, the proposed prototype shows that it is possible to design and implement an efficient sound-by-construction static analyzer based on abstract interpretation for self modifying code.
The implementation has been tested on some significant **eval** patterns (i.e., **eval** usages taken from real-world malware compiled in our core language). In particular, we have considered the 2017 folder of the collection of JavaScript malware provided in [1]. In this section, we consider the subset of malware that uses *explicit* **eval** calls, as discussed in the introduction. These are the real-world malware form which we extracted the **eval** patterns used for testing the analyzer. The process of extracting **eval** patterns has two steps, first we purge the malware code from all the language features not in our language (that, by definition of our language, are not related to **eval**, hence do not affect the execution of **eval**) such as function inline, objects or HTML constructs. Then, we compile the resulting JavaScript code in our language. Fortunately, malware code is usually short (few lines) and therefore it was possible for us to check manually the faithfulness of the extracted **eval** patterns with the original use of **eval** in the real-world malware considered.
From the malware using explicitly **eval**, we have extracted the 20 different **eval** patterns used for testing the analyzer that can be summarized and clustered in the following four categories and that will be discussed in Sect. 5.2 and 5.3: (i) malware manipulating an **eval** input that can be approximated as a finite set of strings, (ii) malware manipulating an **eval** input that can be approximated as an infinite set of strings, (iii) malware using nested **eval** calls, (iv) malware manipulating statically unknown inputs in **eval**.

Before discussing strength and weakness of our analyzer w.r.t. the aforementioned categories, let us show how it works on one of the extracted patterns.

5.1 A case study: Analyzing a real-world malware **eval** pattern

In order to show how of the proposed approach can be exploited, we discuss and analyze an example of **eval** pattern extracted from a common JavaScript malware. A widespread JavaScript malware category is the drive-by-download malware. A typical behavior of these malware, living in malicious or compromised web sites, consists of executing JavaScript code on the victim browser, downloading an executable (e.g., ransomware, virus) from another malicious site and executing it on the victim machine.

In Fig. 16, we report the **eval** pattern of such a JavaScript malware in [1]. The malware goal is to hide and obfuscate the creation of an `ActiveXObject` element, manipulating some variables containing strings and transforming the result in executable code at line 21, by using **eval**. In particular, lines 4-12 manipulate the variable `v` into a loop and save the result into the variable `str`. Then, at line 12, the lower-case value of `str` is concatenated with `=new`, obtaining `d=new`. The variable `d` is the variable to which will be assigned the `ActiveXObject`. Lines 13-20 manipulate the string variable `ss`, in order to extract the string `ActiveXObject` and to save it into the variable `p`. Finally, line 20 put together the partial string results previously obtained and lines 21 execute the string `p`, corresponding to the statement `d = new ActiveXObject(WScript.Shell);`.

Let us analyze this fragment by using our analyzer. The abstract value of the variable `p` at line 21 is the finite state automaton $A_{\mathtt{p}}$ reported in Fig. 17, where we use the short-cut $c \in \mathtt{ss}$ in a single transition to denote any possible transition between two states with a character of the string value of `ss` at line 13. In particular, this transition is repeated for 13 times and this imprecision is due to the fix-point computation at lines 16-18. Similarly, some noise is added also by the fix-point computation at lines 4-10, that it is reflected in the initial transitions of $A_{\mathtt{p}}$, namely the transitions reading the characters `c`, `d` and `f`. Since the automaton $A_{\mathtt{p}}$ is used as abstract input of **eval**, the procedure described in the previous section is activated in order to generate the CFG $\mathtt{G}_{\mathtt{p}}$ approximating the concrete execution of the **eval** call. In particular, $\mathtt{CFGGen}(\mathtt{StmSyn}(A_{\mathtt{p}}))$ is called and the resulting CFG $\mathtt{G}_{\mathtt{p}}$ is reported in Fig. 18, where we have collapsed consecutive **true** edges in a single one for space limitations. The generated CFG contains the concrete execution of **eval** at line 21, namely

```
1   var v = "CDF";
2   i = 0; str = "";
3
4   while (Math.random() != 0.5) {
5     if (str === "D")
6       break;
7     if (i % 2 != 0)
8       str= str + v.charAt(i);
9     i++;
10  }
11
12  str = str.charAt(0).toLowerCase() + "=new";
13  ss = "ACDcPDtASiFDvWEeERXLLOQWbAQjFCeXScASt'PALPSKFdHeFaLDcSHSD";
14  p=""; i=-1;
15
16  while (Math.random() != 0.061 && ++i < 100)
17    if (i % 3 == 0)
18      p= p + ss.charAt(i);
19
20  p = str + p.substring(0, 13) + "(WScript.Shell);";
21  eval(p);
```

Fig. 16. JavaScript malware fragment sample.

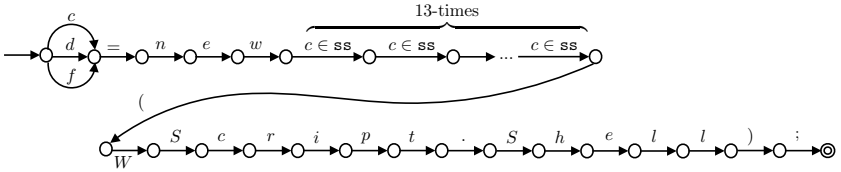


Fig. 17. A_{p} abstracting the value of p at line 21 of Fig. 16.

`d = new ActiveXObject(WScript.Shell);`, showing that a possible malevolent action may be performed, but it also contains other legal statements, such as `c = new ActiveXObject(WScript.Shell);`. In particular, $\mathsf{G_p}$ may execute a single assignment, either to `c`, or `d` or `f`, that calls a constructor whose name length is 13 and its characters are taken from the ones contained in `ss`, with input `WScript.Shell`. As we have already mentioned before, some noise, in the string abstraction, is added during the fix-point computation at lines 16-18 of the program reported in Fig. 16 and this is reflected in $\mathsf{G_p}$. For example, also the legal statement `c = new ObjectXActive(WScript.Shell);` may be executed in the CFG, but, the object `ObjectXActive` is not declared in the fragment hence, will lead to an exception during the CFG execution. Hence, any statement assigning to `c`, `d` or `f` an object `ActiveXObject(WScript.Shell)` will be abstractly executed, while the other statements lead to an exception, since these objects, as previously explained, are not declared in the fragment.[9] Other interesting properties can be derived from the CFG $\mathsf{G_p}$. For example, `G` surely does not contains any other explicit **`eval`** call, helping to tune the nested call widening, as

[9]At the moment, our prototype analyzer does not support exceptions and in this case not executable strings are treated as no-op statements.

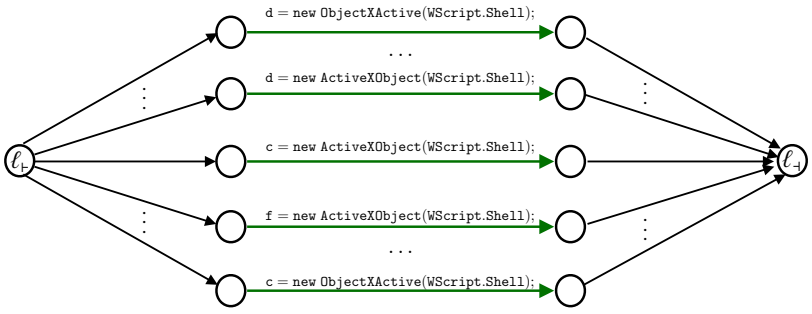


Fig. 18. Control-flow graph G_p corresponding to $\mathtt{CFGGen}(\mathtt{StmSyn}(A_\mathsf{p}))$.

discussed in Sect. 3.3. Moreover, we can state that the only variables c, d and f may be updated by the **eval** call at line 21, while the other will not surely be modified.

5.2 Strengths: What the analysis can do

In this section, we report the most significant strength points of the analysis proposed. In order to measure quality and precision of our analyzer, we tackle the following questions:

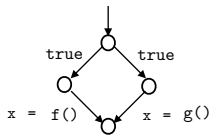
Q1: Does the analyzer handle efficiently string-to-code statements (**eval**), even in presence of join points?
Q2: Does the analyzer handle nested calls to **eval**?

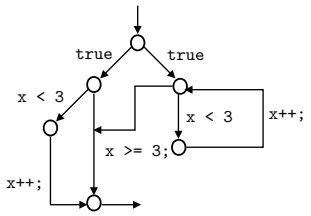
eval *of dynamically-generated strings (**Q1**).* As observed before, the proposed executability string analysis allows the analyzer to handle non-standard uses of **eval**, where the **eval** input string is dynamically manipulated and is not replaceable by equivalent **eval**-free statements [37]. In the following, we describe three representative **eval** examples concerning the first two malware categories discussed at the beginning of Sect. 5.

Consider the code fragment in the first line of Fig. 19 (on the left the code fragment, on the right the code generated by $\mathsf{Exe}^\#(A_{\mathtt{str}})$ just before the **eval** execution). Note that the boolean value of the **if**-guard B is statically unknown hence both the branches must be taken into account. This implies that, the statements executed by **eval** may assign to x either the value returned by the function f or the one returned by the function g. We soundly approximate the code potentially executed by **eval** with the CFG reported in the second column of the first row in Fig. 19 ($\mathsf{Exe}^\#$ output). This is a simple example of non replaceable **eval** usage, because the string str is not constant, that we can analyze. Nevertheless, this is not a really significant example being an *easy case* since we believe that it is possible to find ad-hoc solutions for replacing also this kind of **eval** uses, being, the string manipulation considered here, a simple concatenation of syntactic categories of the language. As far as precision is concerned, in this case the synthesized CFG is precise since it precisely contains the two possible executions.

Let us now consider a more challenging example, the one provided in the second row of Fig. 19. Also in this case, the boolean value of the **if**-guard B is statically unknown, hence **eval** may execute either an **if** or a **while** statement. In this case, the code that will be potentially executed is not a simple combination of syntactic language structures, for this reason we believe this is an harder case to tackle for existing analysis tools. The sound approximation of the potentially executed code is reported in the second row. It is worth noting that

Code	Exe# output

`str = "x=";`
`if (B)`
`  str = str + "f";`
`else`
`  str = str + "g";`
 	
`eval(str + "()");`
	

`if (B)`
`  str = "if";`
`else`
`  str = "while";`
`str = str + "(x<3){x++;}"`
 	
`eval(str);`
	

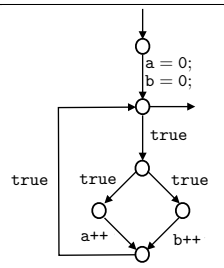
`str = "a=0;b=0;";`
`while (i++ < 100)`
`  if (B)`
`  str = str + "a++;";`
`  else`
`  str = str + "b++;";`
 	
`eval(str);`
	

Fig. 19. **eval** patterns results for **Q1**.

this example does not represent the most challenging possible use of **eval** since its input string is dynamically generated at a *join point* only due to a conditional branch. As before, the synthesized program is precise since it precisely contains the two possible programs to execute.

In the last scenario, we consider the code fragment reported in the last row of Fig. 19, where the string is built in a **while** loop, namely the **eval** input string is approximated after a **while** statement *join point*. In this case, the analysis is complicated by the fact that we have also to approximate (by computing a widening) the **while** loop execution, in order to avoid divergence. Suppose the number of loop iterations is unknown due to the statically unknown value of the variable `i` before the **while** statement, then we have to use a widening operator on automata for ensuring termination (Sect. 3.1). In particular, in the example we use the widening operator ∇_5, allowing us to over-approximate the language for `str` by the automaton recognizing the regular expression `a=0;b=0;(a++ || b++)`*. It is possible to tune string approximation precision, and therefore to obtain different code approximations, by changing the widening operator used in the analysis. The corresponding CFG, over-approximating the code executed by **eval**, is shown in the last row, second column.

*Nested eval calls (**Q2**).* As explained in Sect. 3.3, the soundness and termination of our approach is guaranteed by the nested call widening, i.e., a threshold for the nested call depth, that we denote by $\tilde{\tau}$. It may appear useless to spend effort to tackle situations that are considered rare in real-world code, as indeed it happens in TAJS where this threshold is fixed to 1 [38], but we believe that to ignore this potential situation may be a shortsighted choice.

Indeed, the TAJS authors have in mind only non malicious code, while, as we explained in the introduction, we observed that, in malware, nested **eval** is used in the 10% of samples, hence in malicious code this situation is not so rare and may increase in the future, as the trend, in the last years, shows. Moreover, at least one sample contains a diverging nested **eval** sequence. Hence, it should be clear that, only by spending some effort in handling nested eval calls, we can guarantee termination (in presence of divergent sequences of calls) and a tunable precision, by working on the threshold.
In order to show how the analysis behaves in these situations, let us consider two significant examples of nested **eval** calls: the first example is a terminating sequence of nested **eval** calls, while the second one is a non-terminating sequence. Consider the code fragment below.

```
a=0;
str = "a++;if(a < 3)eval(\"a++;\" + str);";
eval(str);
```

In this example, as long as a is less then 3, the program concatenates the string "a++;" with str, while, when a becomes greater than or equal to 3, the **eval** call returns, closing the sequence of nested calls. It is clear that, the analysis result depends on the threshold of the nested call widening $\tilde{\tau}$. If the nested call widening threshold is greater than or equal to 3, no loss of precision occurs during the analysis, precisely handling the whole sequence of nested **eval** calls. Otherwise, the analysis gives up, returning the $\top$ abstract state (i.e., all the possible program variables evaluated to $\top$) as explained in Sect. 3.3. In this way, while preserving soundness, the analysis may continue on the code after the **eval** call causing the nested call, still able to get potentially significant information about the programs. We believe that this is an important added value of our analyzer, since most of the existing static analysis tools simply get stuck the execution when a non-handled case occurs, returning no useful analysis information. Next code fragment shows an example of non-terminating sequence of nested **eval** calls.

```
a=0;
str = "a++;if(a < 3)str = \"a++\" + str;eval(str);";
eval(str);
```

In this case, independently from the choice of the nested call widening, the static analyzer has to give up because the program diverges, and, in order keep soundness, a $\top$ abstract state is returned.

It is worth noting, that in this context the higher is the threshold and the more we improve precision when we have to deal with deep sequences of nested eval calls but, on the contrary, the later we can stop divergent computations. In this sense, we have to decide a trade-off between precision and costs (in presence of divergence), and this can be found by analyzing which of these situations is statistically more likely in the analysis context, i.e., either a deep nesting degree or divergence due to infinite programs.

5.3 Weaknesses and future ideas: Where we lose precision and how we think we may improve it

The main limitation of static analysis is its source of imprecision. In our analysis there are several sources of imprecision that we have recognized. The first is a general problem of static analysis, i.e., the use of widening for guaranteeing termination, while there are other two situations where imprecision is due to the impossibility of extracting a CFG from the FA in the proposed framework. In the following we discuss both these sources of imprecision and our ideas for tuning/improving the framework.

Widening. The impact of widening on precision is similar to the one discussed for nested **eval** calls, and again necessary for avoiding divergence. As usual in static analysis, precision in presence of widening can be improved by deciding to apply widening only after a fixed number of precise (by using the least upper bound) iterations. Precisely as it happens for nested **eval** calls, the greater is the threshold for starting using the widening, the more precisely we can analyze terminating computations with a high number of loop iterations, but also the later we can start making the fix-point computation faster. Hence, again we have to decide the trade-off between

```
str = "x=";
while (i < 3)
  str = str + "5";
str = str + ";"; eval(str);
```

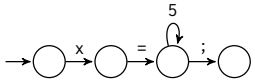


Fig. 20. A s.t. $\mathscr{L}(A) = \{\texttt{x} = 5^n; \mid n \in \mathbb{N}\}$

precision and computational costs in presence of divergent (or very long) computations.
Moreover, as far as the automata widening is concerned, we recall that we can also tune the precision of the widening itself by deciding the length of the words to compare for computing the widening (the higher is this length, the more precise is the operation and the more the operation costs in terms of efficiency, see Sect. 3.1).

Not cycle-executable FA. As we observed in Sect. 4, there are particular forms of FA (which occur when the string is dynamically generated by loops) avoiding the possibility of generating a CFG approximating the code executed by an **eval**. In particular, we observed that, if the **eval** analysis abstracts the **eval** argument with a not cycle-executable FA, then we have to return the $\top$ (unknown) value for all the variables and continue the analysis after **eval**. Nevertheless, statistically speaking, we can still be enough precise for more than the 50% of malware of the analyzed folder that uses explicit **eval** calls . In order to better explain the problem, consider the code fragment reported in Fig. 20, where we assume that the value of `i` is statically unknown. Moreover, consider to apply ∇_2 in the **while**-loop. In Fig. 20, we report the automaton A representing the abstract value of the variable `str` before the **eval** execution, where the cycle in the automaton is caused by the application of the widening ∇_2 to ensure termination. As explained in Sect. 4, the automaton A is not *cycle-executable*, since there exists a cycle not involving a μJS statement. In this case, our analyzer cannot return a CFG over-approximating the code that may be potentially executed since, intuitively, the hypothetical CFG should be infinite for capturing all the possible assignments described by the FA, namely all the assignments of any possible number formed only by 5 to the variable `x` (i.e., `x=5;`,`x=55;`,`x=555;`...). Nevertheless, let us recall that soundness is still satisfied, since the property is decidable, and, when it does not hold, the top abstract state is returned as output of the **eval** execution, allowing the analysis to continue.
In order to improve precision when dealing with these kinds of FA, allowing us to be more precise for a higher number of potential **eval** patterns, we have to find a way to extract CFGs also from not cycle-executable FA. The idea we are currently investigating consists in defining a form of *abstract CFG* able to finitely represent a potential infinite number of CFGs, in the example it would be the CFG for `x=5`*. Unfortunately, things are not so easy as it may seem, since the whole framework has to be generalized in order to construct the analyzer in such a way that it is able to interpret also these forms of abstract CFGs. It is clear that this idea deserves further research but it can really widen the potentialities of our approach.

Statically unknown inputs. An important source of imprecision in our framework is the presence of dynamic/unknown inputs in the computation of the **eval** argument. Indeed, in this case it results that the string abstract domain used does not allow us to keep enough information, on the **eval** argument, when its computation involves unknown inputs. Let us explain the situation by means of an example: Consider the following **eval** pattern

```
eval("if(B){x=x+1;}else{" +  readStr() + "}");
```

where `B` is a statically unknown boolean guard and `readStr()` reads a string input value that, being a statically unknown value, is abstracted to the automaton recognizing any possible string over the alphabet Σ (i.e., single state that is both initial and final with a self-transitions for each $\sigma \in \Sigma$, namely a non cycle-executable automaton). This means that the resulting abstract value of the **eval** argument contains this *top* automaton, and therefore it is, itself, a non cycle-executable automaton. Hence, whenever we have an unknown/dynamic input the analysis returns a

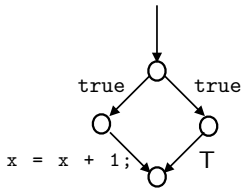


Fig. 21. Example of CFG with T extension

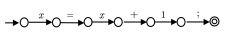
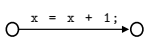
non cycle-executable automata, and therefore, as already discussed above, it is not able to extract a CFG and it must returns the $\top$ state, allowing the analysis to continue after the **eval** statement. As far as the analyzed malware repository is concerned, we observed that the 47% of considered malware using explicit **eval** calls generate non cycle-executable FA due to unknown inputs, representing a relevant source of imprecision to tackle. Our idea for improving precision in these situation consists, first of all, in improving the string abstract domain by embedding into the alphabet Σ a special character T, to be intended as *"any possible string"* (following [50]). Hence, we replace the non cycle-executable FA recognizing any string with a symbol representing it. In this way, the unknown inputs do not generate non cycle-executable FA while keeping in the FA the same information. In the example, the abstract value of the **eval** input would be the automaton recognizing the string `if(B){x=x+1;}else{T}`. Hence, this solution combined with the possibility of generating abstract CFG (described in the previous paragraph) would allow the analysis to return the abstract CFG reported in Fig.21, where the $\top$ label denotes any program. In real world settings, this would allow the analysis to detect at least patterns, in the executed code, potentially dangerous due to the presence of specific statements or strings.

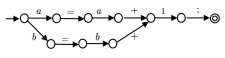
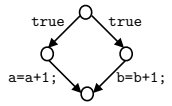
6 RELATED WORK

The aim of the proposed analysis is to extend existing works on static analysis of dynamic languages, by allowing us to analyze also obfuscated (not removable) uses of **eval**. In this way, we are able to analyze also potentially hidden *malicious* behaviours. Hence, the related works are those introducing string analyses, since the choice we made is central in the construction provided. Then, we describe the existing static analyses for dynamic languages (usually considering only removable **eval** or ignoring it) and, in general for self-modifying code (e.g., for other languages such as assembly). Finally, we describe other (dynamic or hybrid) approaches for *detecting* (not in general analyzing) malware potentially using **eval**.

String analysis. The analysis of strings is nowadays a relatively common practice in program analysis due to the widespread use of dynamic scripting languages. Examples of analyses for string manipulation are in [16, 26, 41, 48, 56, 62]. In particular, abstract parsing w.r.t. the programming language grammar for abstracting strings has been proposed in [26], tracking executable strings for each program variable, but it loses any other kind of information on strings properties in general (i.e., not executable strings).
The use of symbolic (grammar-based) objects in abstract domains is also not new (see [21, 35, 57]) and some works explicitly use transducers for string analysis in script sanitisation ([36] and [62]), all recognizing that specifying the analysis in terms of abstract interpretation makes it suitable to potential combinations with other analyses, providing a better potential in tuning accuracy and costs. None of these works use string analysis for analyzing executability of dynamically generated code.

Static (and dynamic) analyses (of not-malicious code). One of the most complete static analyzers, available for JavaScript, based on abstract interpretation and that handles **eval**, is TAJS [37, 38, 49], for this reason we provide

P	TAJS result	String analysis of y	Analysis of y
`y="x=x+1;"; eval(y);`	`x=x+1;`		

`if (x > 0)`
`  y="a=a+1;";`
`if (x < 0)`
`  y="b=b+1;";`
`eval(y);`
	Analysis Limitation Exception		

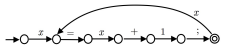
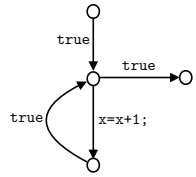
`y= "";`
`while (x < 3) {`
`  y = y + "x=x+1;"; x=x+1;`
`}`
`eval(y);`
	Analysis Limitation Exception		

Fig. 22. Comparison with TAJS

a deeper comparison with this work. In [37], the authors introduce an automatic code rewriting techniques removing **eval** constructs in JavaScript applications. This work has been inspired by the work of Richards et al. [53] showing that **eval** is widely used. Nevertheless, in many cases, its use can be simply replaced by JavaScript code without **eval**. In particular, the authors integrate a refactoring of the calls to **eval** into the TAJS data-flow analyzer. TAJS performs inter-procedural data-flow analysis on an abstract domain of objects capturing whether expressions evaluate to constant values. In this case, **eval** calls can be replaced with alternative code that does not use **eval**. It is clear that code refactoring is possible only when the abstract analysis recognizes that the arguments of the **eval** call are constants. Moreover, they handle the presence of nested **eval** by fixing a maximal degree of nesting, but in practice they set this degree to 1. Instead, the solution we propose allows us to go beyond constant values and refactor code also when the arguments of **eval** are elements of a regular language of strings.
We compare our analyzer with TAJS (version 0.9-8), considering three **eval** pattern, which allow us to underline the differences between TAJS and our prototype. We report three significant examples (one for each one of these classes) in Fig. 22, where we summarize the comparison with TAJS. The first class of tests consists of all the programs where the string variables collect only one value during execution, i.e., they are constant string variables. The second class of tests consists of all the programs where there are no constant string variables, namely variables whose value before the **eval** call is not precisely known and it is approximated by a finite set of potential string values. The last class of examples (third row) is similar to the previous one, except that the **eval** input string values cannot be approximated by a *finite* set, namely `x=x+1;(x=x+1;)`*. What we can observe is that, in the second and in the third class of programs, TAJS loses the value of `y`, while, in all the cases, our analyzer performs a sound over-approximation of the set of values computed in `y`, even if with some degree of imprecision (as discussed in the previous paragraph)[10].

Other static analyzers for JavaScript, based on abstract interpretation, have been developed, such as JSAI [40] and SAFE [42, 52]. They look for a flexible, configurable and tunable tool focusing on context-sensitiveness, heap-sensitiveness [40] and loop-sensitiveness [52]. Nevertheless, they do not explicitly mention solutions to dynamic

[10]The computed automata are reported in the third column, while the CFGs obtained from the abstract value of `y` are reported in the fourth column.

generated code by **eval**. TamiFlex [12] also synthesizes a program at every **eval** call by considering the code that has been executed during some (dynamically) observed execution traces. The static analysis can then proceed with the so obtained code without **eval**. It is sound only with respect to the considered execution traces, producing a warning otherwise. Finally, a static analysis for a static subset of PHP (which ignores **eval**-like primitives) has been developed in [11]. Staged information flow for JavaScript in [17] with *holes* provides a conditional (a la abduction analysis in [30]) static analysis of dynamically evaluated code. Symbolic execution-based static analyses have been developed for scripting languages, e.g., PHP, including primitives for code reflection, still at the price of introducing false negatives [60]. Similarly to TAJS, [47] proposes a semi-automatic *dynamic* technique for replacing benevolent **eval** patterns with an equivalent sequence of statements without **eval**, validating their approach of benevolent **eval** use cases.

Static analysis of self-modifying code. We are not aware of effective general purpose sound static analyses handling self-modifying code for high-level scripting languages. On the contrary, a huge effort was devoted to bring static type inference to object-oriented dynamic languages (e.g., see [2] for an account in Ruby) but with a different perspective: *Bring into dynamic languages the benefits of static ones – well-typed programs don't go wrong*. Our approach is different: *Bring into static analysis the possibility of handling dynamically mutating code*. The authors of [3] have a similar approach for analyzing self-modifying code for an assembly language, modeling code as state-enhanced CFGs. Differently from us, they perform a dynamic analysis for extracting a code representation which is descriptive enough to include most code mutations by a dynamic analysis, and then reform analysis on a linearization of this code. On the semantics side, since the pioneering work on certifying self-modifying code in [15], the approach to self-modifying code consists of treating machine instructions as regular mutable data structures, and to incorporate a logic dealing with code mutation within a la Hoare logics for program verification.
Static *taint analysis* keeping track of values derived from user inputs has been developed for self-modifying code by partial derivation of the CFG [59]. The approach is limited to taint analysis, e.g., to limit code-injection attacks.

Malware dynamic and hybrid analyses. In the literature, there is a huge amount of work on dynamic and hybrid (static and dynamic) analyses for detecting malware.
Concerning dynamic analysis, a big effort had been spent to detect injection attacks in the JavaScript world by means of taint analysis. It is worth noting that, these approaches are in some way complementary to ours. In [29] the authors detect injection attacks in Node.js applications. In the same context, the authors of [55] propose a hybrid approach for injection attacks detections, integrating a static analysis to approximate the set of commands reaching a sink (e.g., **eval** or `exec` functions) and a dynamic analysis to track strings reaching those sinks. Finally, the authors of [39] aims to a platform-independent injection analysis, based on dynamic taint analysis, in order to be instantiated to the several existing JavaScript engines. Concluding, all these works perform an analysis of how information flows from inputs to **eval** statements in order to detect potential injection attacks, but they do not deepen the problem of understanding whether the **eval** will execute something dangerous. On the other hand, we do not analyze from where the potential attack comes from, but we extract what the **eval** potentially executes, and therefore we can detect whether what is executed may be dangerous. In other words, they can answer to the question "is the source of executed statements trusted?", while we can answer to the question "May be the executed statement dangerous?". Surely, it could be very interesting to combine these two approaches for exploiting dynamic analysis in order to make static analysis more precise (for instance, we could analyze executability for detecting what may be executed only when the source is untrusted).
Finally, there are some hybrid approaches to detect JavaScript malware. In [23] the authors propose a browser extension for JavaScript malware detection, base on Bayesian classification of hierarchical features of the JavaScript abstract syntax tree to identify syntax elements that are highly predictive of malware. In other words, it differs from our approach since we statically analyze the entire program *semantics*, while they statically analyze the

syntax structure of what will be executed, by hooking the runtime just before it is executed. [58] mixes machine learning and dynamic analysis techniques to provide an hybrid approach for JavaScript malware classification and detection. In the proposed approach the static part of the technique is based on machine learning-based extraction of syntactic and behavioral features, and therefore again the static component is more a syntactic analysis, while the semantic analysis is performed dynamically. [61] proposes an automatic behavior model for malware detection and classification. In particular, the authors model the malware system call graph from its dynamic execution traces as a FA and propose a learning framework for the classification of 8 different types of popular JavaScript attacks. Again the semantic analysis is dynamic, and the resulting model is statically classified.

7 CONCLUDING DISCUSSION

We conclude highlighting the value, in the context of static analysis, of the framework presented in this paper. The analysis we have proposed attacks an extremely hard problem in static program analysis by abstract interpretation, since the standard static analysis assumption (i.e., the program code we want to analyze must be static) is broken when we have to deal with string-to-code statements. In this paper, we have shown that even without this assumption, it is still possible for static analysis to semantically analyze dynamically mutating code in a meaningful and sound way. This provides the very first proof of concept sound static analysis for self-modifying code based on bounded reflection for a high-level script-like programming language. Hence, our main original contribution consists in proposing an innovative approach for designing sound static analyzers for dynamic code, consisting in recursively calling the abstract interpreter on an approximation of the dynamically generated code. Once the recursive call returns, we continue the standard analysis. The approach we propose is, in this sense, a truly *dynamic static analyzer*, keeping the analysis going on, even when code is dynamically built.

Clearly, the framework proposed is just the beginning and still there is much work to do. The most important future work consists in improving precision, as we discussed in Sect. 5.3, by allowing the analysis to analyze *abstract CFGs* and by extending the string abstract domain for representing also $\top$ automaton [50]. Another direction for improving precision can be that of integrating the proposed static analysis in an hybrid solution, by using, for instance, taint analysis (or other dynamic analyses) for driving when to apply static analysis, as briefly discussed in Sect. 6. Finally, we have considered only **`eval`** as string-to-code statement, while there are other ways, for dynamically executing code built out of strings, that should be investigated. However, we strongly believe that, the same approach used for **`eval`**, could be easily applied to any other string-to-code statement.

From a more theoretical point of view, interesting future works consist in exploiting the proposed approach for analyzing code, independently from the presence of string-to-code primitives, in order to investigate, on dynamic languages, several application contexts where static analysis by abstract interpretations has been exploited. First of all, we could trace (abstract) flows of information during execution [32, 34, 44, 45] in order to tackle different security issues, such as the detection of (abstract) code injections [9, 14] or the formal characterization of dynamic code obfuscators and of their potency [31, 33]. Moreover, the ability to analyze malware code could be exploited for extracting code properties which could be used for analyzing code similarity [24], a technique useful for instance to identify or at least classify malicious code.

REFERENCES

[1] Hynek petrak js malware collection. https://github.com/HynekPetrak/javascript-malware-collection.

[2] An, J. D., Chaudhuri, A., Foster, J. S., and Hicks, M. Dynamic inference of static types for Ruby. In *POPL* (2011), T. Ball and M. Sagiv, Eds., ACM, pp. 459–472.

[3] Anckaert, B., Madou, M., and Bosschere, K. D. A model for self-modifying code. In *Information Hiding* (2006), J. Camenisch, C. S. Collberg, N. F. Johnson, and P. Sallee, Eds., vol. 4437 of *LNCS*, Springer, pp. 232–248.

[4] Arceri, V., and Maffeis, S. Abstract domains for type juggling. *Electr. Notes Theor. Comput. Sci. 331* (2017), 41–55.

[5] Arceri, V., and Mastroeni, I. An automata-based abstract semantics for string manipulation languages. In *Proceedings Seventh International Workshop on Verification and Program Transformation, VPT@Programming 2019, Genova, Italy, 2nd April 2019* (2019),

pp. 19–33.

[6] Arceri, V., and Mastroeni, I. A sound abstract interpreter for dynamic code. In *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020* (2020), C. Hung, T. Cerný, D. Shin, and A. Bechini, Eds., ACM, pp. 1979–1988.

[7] Arceri, V., Mastroeni, I., and Xu, S. Static analysis for ecmascript string manipulation programs. *Appl. Sci. 10* (2020), 3525.

[8] Arceri, V., Olliaro, M., Cortesi, A., and Mastroeni, I. Completeness of abstract domains for string analysis of javascript programs. In *Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings* (2019), R. M. Hierons and M. Mosbah, Eds., vol. 11884 of *Lecture Notes in Computer Science*, Springer, pp. 255–272.

[9] Balliu, M., and Mastroeni, I. A weakest precondition approach to robustness. *Trans. Comput. Sci. 10* (2010), 261–297.

[10] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Gros, C., Kamsky, A., McPeak, S., and Engler, D. R. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (2010), 66–75.

[11] Biggar, P., and Gregg, D. Static analysis of dynamic scripting languages. Technical report, Department of Computer Science, Trinity College Dublin, 2009.

[12] Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., and Mezini, M. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proc. of the 33rd Internat. Conf. on Software Engineering, ICSE 2011* (2011), pp. 241–250.

[13] Brzozowski, J. A. Derivatives of regular expressions. *J. ACM 11*, 4 (1964), 481–494.

[14] Buro, S., and Mastroeni, I. Abstract code injection - A semantic approach based on abstract non-interference. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings* (2018), I. Dillig and J. Palsberg, Eds., vol. 10747 of *Lecture Notes in Computer Science*, Springer, pp. 116–137.

[15] Cai, H., Shao, Z., and Vaynberg, A. Certified self-modifying code. In *PLDI* (2007), J. Ferrante and K. S. McKinley, Eds., ACM, pp. 66–77.

[16] Christensen, A. S., Møller, A., and Schwartzbach, M. I. Precise analysis of string expressions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings* (2003), R. Cousot, Ed., vol. 2694 of *Lecture Notes in Computer Science*, Springer, pp. 1–18.

[17] Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged information flow for JavaScript. In *PLDI* (2009), M. Hind and A. Diwan, Eds., ACM, pp. 50–62.

[18] Cousot, P. Types as abstract interpretations (invited paper). In *Conference Record of the 24th ACM Symposium on Principles of Programming Languages* (*POPL '97*) (1997), ACM Press, pp. 316–331.

[19] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages* (*POPL '77*) (1977), ACM Press, pp. 238–252.

[20] Cousot, P., and Cousot, R. Abstract interpretation frameworks. *J. Logic and Comput. 2*, 4 (1992), 511–547.

[21] Cousot, P., and Cousot, R. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture* (25–28 June 1995), ACM Press, New York, NY, pp. 170–181.

[22] Cousot, P., and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1978), ACM Press, pp. 84–96.

[23] Curtsinger, C., Livshits, B., Zorn, B. G., and Seifert, C. ZOZZLE: fast and precise in-browser javascript malware detection. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings* (2011), USENIX Association.

[24] Dalla Preda, M., Giacobazzi, R., Lakhotia, A., and Mastroeni, I. Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. *ACM SIGPLAN Notices 50*, 1 (2015), 329–341.

[25] Davis, M., Sigal, R., and Weyuker, E. J. *Computability, Complexity, and Languages, Second Edition: Fundamentals of Theoretical Computer Science (Computer Science and Scientific Computing) 2nd edition.* Elsevier, 1994.

[26] Doh, K., Kim, H., and Schmidt, D. A. Abstract parsing: Static analysis of dynamically generated string output using lr-parsing technology. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings* (2009), J. Palsberg and Z. Su, Eds., vol. 5673 of *Lecture Notes in Computer Science*, Springer, pp. 256–272.

[27] Drape, S., Thomborson, C., and Majumdar, A. Specifying imperative data obfuscations. In *ISC - Information Security* (2007), J. A. Garay, A. K. Lenstra, M. Mambo, and R. Peralta, Eds., vol. 4779 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 299 – 314.

[28] D'Silva, V. Widening for automata. Diploma Thesis, Institut Fur Informatick, Universitat Zurich, 2006.

[29] Gauthier, F., Hassanshahi, B., and Jordan, A. AFFOGATO: runtime detection of injection attacks for node.js. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018* (2018), J. Dolby, W. G. J. Halfond, and A. Mishra, Eds., ACM, pp. 94–99.

[30] Giacobazzi, R. Abductive analysis of modular logic programs. *J. of Logic and Computation 8*, 4 (1998), 457–484.

[31] Giacobazzi, R., Jones, N. D., and Mastroeni, I. Obfuscation by partial evaluation of distorted interpreters. In *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'12)* (2012), O. Kiselyov and S. Thompson, Eds., ACM Press, pp. 63 – 72.

[32] Giacobazzi, R., and Mastroeni, I. A proof system for abstract non-interference. *J. Log. Comput. 20*, 2 (2010), 449–479.
[33] Giacobazzi, R., and Mastroeni, I. Making abstract interpretation incomplete: Modeling the potency of obfuscation. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings* (2012), A. Miné and D. Schmidt, Eds., vol. 7460 of *Lecture Notes in Computer Science*, Springer, pp. 129–145.
[34] Giacobazzi, R., and Mastroeni, I. Abstract non-interference: A unifying framework for weakening information-flow. *ACM Trans. Priv. Secur. 21*, 2 (2018), 9:1–9:31.
[35] Heintze, N., and Jaffar, J. Set constraints and set-based analysis. In *Principles and Practice of Constraint Programming, Second International Workshop, PPCP'94, Rosario, Orcas Island, Washington, USA, May 2-4, 1994, Proceedings* (1994), A. Borning, Ed., vol. 874 of *Lecture Notes in Computer Science*, Springer, pp. 281–298.
[36] Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., and Veanes, M. Fast and precise sanitizer analysis with BEK. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings* (2011), USENIX Association.
[37] Jensen, S. H., Jonsson, P. A., and Møller, A. Remedying the eval that men do. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012* (2012), M. P. E. Heimdahl and Z. Su, Eds., ACM, pp. 34–44.
[38] Jensen, S. H., Møller, A., and Thiemann, P. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings* (2009), pp. 238–255.
[39] Karim, R., Tip, F., Sochurkova, A., and Sen, K. Platform-independent dynamic taint analysis for javascript. *IEEE Transactions on Software Engineering* (2018), 1–1.
[40] Kashyap, V., Dewey, K., Kuefner, E. A., Wagner, J., Gibbons, K., Sarracino, J., Wiedermann, B., and Hardekopf, B. JSAI: a static analysis platform for javascript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014* (2014), pp. 121–132.
[41] Kim, H., Doh, K., and Schmidt, D. A. Static validation of dynamically generated HTML documents based on abstract parsing and semantic processing. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings* (2013), F. Logozzo and M. Fähndrich, Eds., vol. 7935 of *Lecture Notes in Computer Science*, Springer, pp. 194–214.
[42] Lee, H., Won, S., Jin, J., Cho, J., and Ryu, S. Safe: Formal specification and implementation of a scalable analysis framework for ecmascript. In *2012 International Workshop on Foundations of Object-Oriented Languages, ACM* (2012).
[43] Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B. E., Guyer, S. Z., Khedker, U. P., Møller, A., and Vardoulakis, D. In defense of soundiness: a manifesto. *Commun. ACM 58*, 2 (2015), 44–46.
[44] Mastroeni, I., and Nikolic, D. Abstract program slicing: From theory towards an implementation. In *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings* (2010), J. S. Dong and H. Zhu, Eds., vol. 6447 of *Lecture Notes in Computer Science*, Springer, pp. 452–467.
[45] Mastroeni, I., and Zanardini, D. Abstract program slicing: An abstract interpretation-based approach to program slicing. *ACM Trans. Comput. Log. 18*, 1 (2017), 7:1–7:58.
[46] Mavrogiannopoulos, N., Kisserli, N., and Preneel, B. A taxonomy of self-modifying code for obfuscation. *Computers & Security 30*, 8 (2011), 679–691.
[47] Meawad, F., Richards, G., Morandat, F., and Vitek, J. Eval begone!: semi-automated removal of eval from javascript programs. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012* (2012), G. T. Leavens and M. B. Dwyer, Eds., ACM, pp. 607–620.
[48] Minamide, Y. Static approximation of dynamically generated web pages. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005* (2005), A. Ellis and T. Hagino, Eds., ACM, pp. 432–441.
[49] Møller, A. Static Analysis of JavaScript. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Invited talk* (2015).
[50] Negrini, L., Arceri, V., Ferrara, P., and Cortesi, A. Twinning automata and regular expressions for string static analysis, 2020.
[51] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of program analysis.* Springer, 1999.
[52] Park, C., and Ryu, S. Scalable and precise static analysis of javascript applications via loop-sensitivity. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic* (2015), pp. 735–756.
[53] Richards, G., Hammer, C., Burg, B., and Vitek, J. The eval that men do - A large-scale study of the use of eval in javascript applications. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings* (2011), M. Mezini, Ed., vol. 6813 of *Lecture Notes in Computer Science*, Springer, pp. 52–78.
[54] Seidl, H., Wilhelm, R., and Hack, S. *Compiler Design - Analysis and Transformation.* Springer, 2012.
[55] Staicu, C., Pradel, M., and Livshits, B. SYNODE: understanding and automatically preventing injection attacks on NODE.JS. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018* (2018), The Internet Society.
[56] Thiemann, P. Grammar-based analysis of string expressions. In *Proceedings of TLDI'05: 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Long Beach, CA, USA, January 10, 2005* (2005), J. G. Morrisett and M. Fähndrich, Eds., ACM, pp. 59–70.

[57] Venet, A. Automatic analysis of pointer aliasing for untyped programs. *Sci. Comput. Program. 35*, 2 (1999), 223–248.
[58] Wang, J., Xue, Y., Liu, Y., and Tan, T. H. JSDC: A hybrid approach for javascript malware detection and classification. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015* (2015), F. Bao, S. Miller, J. Zhou, and G. Ahn, Eds., ACM, pp. 109–120.
[59] Wang, X., Jhi, Y., Zhu, S., and Liu, P. Still: Exploit code detection via static taint and initialization analyses. In *ACSAC* (2008), IEEE Computer Society, pp. 289–298.
[60] Xie, Y., and Aiken, A. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006* (2006), A. D. Keromytis, Ed., USENIX Association.
[61] Xue, Y., Wang, J., Liu, Y., Xiao, H., Sun, J., and Chandramohan, M. Detection and classification of malicious javascript via attack behavior modelling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015* (2015), M. Young and T. Xie, Eds., ACM, pp. 48–59.
[62] Yu, F., Alkhalaf, M., and Bultan, T. Patching vulnerabilities with sanitization synthesis. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011* (2011), R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds., ACM, pp. 251–260.

A PROOFS

Here we report all the missing proofs of the results presented in this paper, listed in order of appearance.

Proof of Lemma 4.1. This lemma proves that any μJS program can be written as a sequence of partial statements in $\Sigma_{\texttt{pStm}}$. Formally, we need to prove that $\forall \texttt{P} \in \mu\text{JS} \ .\exists k \in \mathbb{N} \ .\{\sigma_i\}_{i\in[0,k]} \in \Sigma_{\texttt{pStm}} \ .\texttt{P} = \mathcal{S}(\sigma_0\sigma_1 \dots \sigma_k)$.

The proof is done by structural induction on the structure of μJS programs. It is worth noting that by definition of μJS syntax, P = c;, and all the statements composing c are separated by a semi-colon (i.e., ;), hence by trivial induction on c we can prove that any statement generated by the grammar μJS is in c followed by a semi-colon. Hence, in the following in the base of the structural induction we consider also c; (we need also c for the induction in the **while** and **if** body). We make the proof by structural induction on c, recalling that $\mathcal{S}$ converts a string of strings (in $(\Sigma^*)^*$) in a string of chars (in Σ^*).

Base cases:

- **skip** and **skip**;: since both **skip**, **skip**; $\in \Sigma_{\texttt{pStm}}$, by definition and since $\mathcal{S}$ on sequences already in Σ^* is the identity, the thesis trivially hold.
- x:=e and x:=e;. Let us prove by cases on e
  - If e does not contain punctuation symbols, both x:=e, x:=e; $\in \Sigma_{\texttt{pStm}}$ and therefore, as in the previous case, we have the thesis.
  - If e contains at least one punctuation symbol. Let $\sigma^{\texttt{e}} \in \Sigma^*$ be the string counterpart of the expression e and $n \in \mathbb{N}$ be its length. Let $k \in \mathbb{N} \setminus \{0\}$ be the number of punctuation symbols occurring in $\sigma^{\texttt{e}}$ and $\{p_i\}_{i\in[0,k-1]}$ their positions in the string $\sigma^{\texttt{e}}$. A representation of the string in $\Sigma_{\texttt{pStm}}$ is
$$\underbrace{\texttt{x:=}\sigma^{\texttt{e}}_0 \dots \sigma^{\texttt{e}}_{p_0}}_{\in \Sigma_{\texttt{Syn}}} \overbrace{\dots \sigma^{\texttt{e}}_{p_1}}^{\in \Sigma_{\texttt{Syn}}} \underbrace{\dots \sigma^{\texttt{e}}_{p_k}}_{\in \Sigma_{\texttt{Syn}}} \overbrace{\dots \sigma^{\texttt{e}}_{n}}^{\in \Sigma_{\texttt{Syn}}}}$$
    Since p_0 is the first position where a punctuation symbol occurs in $\mathcal{S}$(x:=e), the prefix up to $\sigma^{\texttt{e}}_{p_0}$ form a partial statement. The following sub-strings, i.e., all the sub-strings between p_i and p_{i+1}, $i > 0$, form partial statements, since between p_i and p_{i+1}, in $\sigma^{\texttt{e}}$, does not occur any punctuation symbol by construction. The substring of $\sigma^{\texttt{e}}$ after $p_k + 1$ (i.e., the next character after the last punctuation symbol of $\sigma^{\texttt{e}}$ if present) is a partial statement, since it does not contain other punctuation symbols always by construction.
- **eval**(s) and **eval**(s);. Let us prove by cases on s

– s does not contain punctuation symbols. Thus, **eval**(s) = $\mathcal{S}(\sigma_0\sigma_1)$ with $\{\sigma_i\}_{i\in[0,1]} \subseteq \Sigma_{\texttt{pStm}}$ are σ_0 = "**eval**(" and σ_1 = "s)". Instead, **eval**(s); = $\mathcal{S}(\sigma_0\sigma_1\sigma_2)$ with $\{\sigma_i\}_{i\in[0,2]} \subseteq \Sigma_{\texttt{pStm}}$ are σ_0 = "**eval**(", σ_1 = "s)" and σ_2 = ";".
– s contains at least a punctuation symbol. In this case, the string expression s is split similarly to the expression e in the above proof of the assignment case.

Inductive step:

- $c_1; c_2$ (with c_1 not ending with ;). For inductive hypothesis, c_1 and c_2 can be written as concatenation of partial statements, i.e., $\exists k_1, k_2 \in \mathbb{N} \setminus \{0\}$.$\{\sigma_i\}_{i\in[0,k_1]} \subseteq \Sigma_{\texttt{pStm}}$, $\{\delta_i\}_{i\in[0,k_2]} \subseteq \Sigma_{\texttt{pStm}}$.$c_2 = \mathcal{S}(\sigma_0 \dots \sigma_{k_1})$, $c_2 = \mathcal{S}(\delta_0 \dots \delta_{k_2})$. Moreover, by definition ; $\in \Sigma_{\texttt{pStm}}$ hence, we are able to rewrite the statement $c_1; c_2$ with the partial statements $\{\sigma_i\}_{i\in[0,k_1]}$ of c_1, ; and $\{\delta_i\}_{i\in[0,k_2]}$ of c_2.

$$\underbrace{c_1;}_{\sigma_0 \dots \sigma'_{k_1} \in \Sigma^*_{\texttt{pStm}}} \quad \overbrace{\delta_0 \dots \delta_{k_2} \in \Sigma^*_{\texttt{pStm}}}^{c_2}$$

  where $\sigma'_k \triangleq \sigma_k$; which is a partial statement since, by hypothesis c_1 was not ending with a ;.
- **while** (b) {c} and **while** (b) {c}; (with c not ending with ;). Let us prove by cases on b
  – b does not contain punctuation symbols. By inductive hypothesis, the statement c can be rewritten as sequence of partial statements i.e., $\exists k \in \mathbb{N} \setminus \{0\}$.$\{\sigma_i\}_{i\in[0,k]} \subseteq \Sigma_{\texttt{pStm}}$. $c = \mathcal{S}(\sigma_0 \dots \sigma_k)$. Hence, we can compose the **while** statement in the following way (consider the case ending with a semi-colon ;, the other one is similar without the last partial statement).

$$\underbrace{\texttt{while(}}_{\in \Sigma_{\texttt{pStm}}} \overbrace{\texttt{b)}}^{\in \Sigma_{\texttt{pStm}}} \underbrace{\texttt{\{}}_{\in \Sigma_{\texttt{pStm}}} \overbrace{\texttt{c\}}}^{\sigma_0 \dots \sigma'_k \in \Sigma^*_{\texttt{pStm}}} \underbrace{\texttt{;}}_{\in \Sigma_{\texttt{pStm}}}$$

    where $\sigma'_k \triangleq \sigma_k$} which is a partial statement since by hypothesis c does not end with a ;.
  – b contains at least a punctuation symbol. In this case, b is split similarly to e in the proof of the assignment.
- **if**(b){c}**else**{c} and **if**(b){c}**else**{c}; (with c not ending with ;). The proof is analogous to the **while** case.

□

Proof of Lemma 4.2. Given an automaton $A = (Q^A, \delta^A, q_0, F^A, \Sigma)$ over the alphabet Σ and the corresponding automaton $A^{\texttt{pStm}} = \texttt{StmSyn}(A)$ over the alphabet $\Sigma_{\texttt{pStm}}$, we generalize the lemma on the language recognized on a generic state $q \in Q^A$, namely $\forall\sigma \in \Sigma^*_{\texttt{pStm}}, \exists q \in Q^A.\ \mathcal{S}(\sigma) \in \mathscr{L}_q(A) \Rightarrow \sigma \in \mathscr{L}_q(A^{\texttt{pStm}})$

It is worth noting that the transformation from A to $A^{\texttt{pStm}}$ is performed by the procedure `Build` (Alg. 3), computing, given $q \in Q^A$, the set I_q of pairs (partial statement, reached state), namely the partial statements recognized from q and the corresponding reached state. The procedure `StmSyn` (Alg. 2) does not affect the set I_q, since it simply builds the desired automaton adding the partial statements and the corresponding reached states to $A^{\texttt{pStm}}$ (lines 6-8). In particular, the procedure stmsyntr, and in turn the procedure `Build` at line 6 of Alg. 2, is recursively called on these reached states. Once a generic couple (σ, q') is added to I_q, the corresponding transition (q, σ, q') is added to $A^{\texttt{pStm}}$. Hence, it is clear that $\exists q' \in Q^A.\ (\sigma, q') \in I_q \Rightarrow \sigma \in \mathscr{L}_q(A^{\texttt{pStm}})$, for some $q' \in Q^A$. For this reason, in order to prove the lemma, we focus on showing, given $q \in Q^A$, that if $\exists q' \in Q^A.\ \mathcal{S}(\sigma) \in \mathscr{L}_q(A)$ then $(\sigma, q') \in I_q$. The proof is conduced by induction on the length of σ.

Base cases: $|\sigma| = 1$, i.e., $\sigma \in \Sigma_{\texttt{pStm}}$. Let suppose that $\exists q \in Q^A.\ \mathcal{S}(\sigma) \in \mathscr{L}_q(A)$. We can split the base cases as follows.

- $\sigma \in$ Punct: The first call to `Build` (line 6 of Alg. 2) is `Build`(A, q_0), that calls buildtr$(q_0, \varepsilon, \emptyset)$ (line 3 of Alg. 3). By definition, any character of Punct is a single character, meaning that there exists a transition in A from q_0 to q labeled with $\mathcal{S}(\sigma)$, namely $(q_0, \mathcal{S}(\sigma), q) \in \delta^A$. The transition is taken into account at line 6

(i.e., the pair $(\mathcal{S}(\sigma), q) \in \Delta_{q_0}$) and will be eventually selected at line 8. No states have been already marked, being the first call to BUILDTR, hence the test at line 9 passes. The test at line 10 fails, since $\sigma \in$ Punct while the test at line 13 is successful, since $\mathcal{S}(\sigma) \in$ Punct and word.$(\mathcal{S}(\sigma)) = \mathcal{S}(\sigma) \in$ Punct $\subseteq \Sigma_{\texttt{pStm}}$. Hence, the couple (σ, q) is added to I_q. Other transitions may be selected at line 8 of Alg. 3, without removing the ones already added.

- $\sigma \notin$ Punct: By definition of $\Sigma_{\texttt{pStm}}$, any single partial statement is a sequence of some non-punctuation symbols ending with a punctuation symbol (e.g., `x :=5;`). Hence, $\mathcal{S}(\sigma)$ can be rewritten as $\mathcal{S}(\sigma) = \mathcal{S}(\sigma' p)$, $p \in$ Punct, $\sigma' \in (\Sigma \setminus \text{Punct})^*$. Let $n = |\mathcal{S}(\sigma')|$ be the length of $\mathcal{S}(\sigma')$ and c_i be the i-th character of $\mathcal{S}(\sigma')$. Since $\mathcal{S}(\sigma) \in \mathscr{L}_q(A)$, there exists a path of A from q_0 to q that reads $\mathcal{S}(\sigma) = \mathcal{S}(\sigma' p)$, that is $\forall i \in [0, n-1]. (q_i, c_i, q_{i+1}) \in \delta^A \wedge (q_n, p, q) \in \delta^A$.
  As in the previous case, the first call to `Build` is `Build`(A, q_0), that calls BUILDTR$(q_0, \varepsilon, \varnothing)$ (line 3 of Alg. 3). The transition (q_0, c_0, q_1), i.e., the transition that reads the first symbol of $\mathcal{S}(\sigma)$, is taken into account at line 6 and will be eventually selected at line 8. Since no states have been marked yet, the test at line 9 is successful. c_0 is not a punctuation symbol, hence, the current call also passes the test at line 10, performing a recursive call to BUILDTR, namely BUILDTR$(q_1, c_0, \{q_0, q_1\})$, accumulating c_0 in the second parameter of the recursive call and searching for the first punctuation symbol. In particular, for any $i \in [0, |n-1|]$, when the transition (q_i, c_i, q_{i+1}) is selected at line 8, any recursive call to BUILDTR on a state q_i will fall down in a recursive call to q_{i+1}, at lines 10-11 of Alg. 3 and it accumulates the current character, namely c_i. Hence, when BUILDTR is called on the state q_n, the parameter word is $\mathcal{S}(\sigma')$, and this call corresponds to the last recursive call of the path we are considering, namely BUILDTR$(q_n, \mathcal{S}(\sigma'), \{q_0, q_1, \ldots, q_n\})$. The transition (q_n, p, q) is added to Δ_{q_n} at line 6 and it will be eventually selected at line 8. Since $p \in$ Punct and $\sigma' p \in \Sigma_{\texttt{pStm}}$, the pair $(\sigma' p, q) = (\sigma, q)$ is added to I_{q_n}.

Inductive step: Let A be a cycle-executable FA and $n \in \mathbb{N}, n > 1$. We suppose that $\forall \sigma \in \Sigma^*_{\texttt{pStm}}. |\sigma| < n, \exists q \in Q^A. \mathcal{S}(\sigma) \in \mathscr{L}_q(A) \Rightarrow \sigma \in \mathscr{L}_q(A^{\texttt{pStm}})$. We prove that $\forall \delta \in \Sigma^*_{\texttt{pStm}}. |\delta| \geq n, \exists q' \in Q^A. \mathcal{S}(\delta) \in \mathscr{L}_{q'}(A) \Rightarrow \delta \in \mathscr{L}_{q'}(A^{\texttt{pStm}})$. Consider $\delta = \sigma\sigma'$, where $\sigma' \in \Sigma_{\texttt{pStm}}$, and suppose that $\exists q' \in Q^A. \mathcal{S}(\delta) \in \mathscr{L}_{q'}(A)$, meaning that there exists a path in A from q_0 to some state q' that reads δ. Since $\mathcal{S}(\sigma)$ is a prefix of $\mathcal{S}(\delta)$, it is clear that, starting from q_0, it will reach some state q of A, namely $\exists q \in Q^A. \mathcal{S}(\sigma) \in \mathscr{L}_q(A)$. Hence, for inductive hypothesis, $\sigma \in \mathscr{L}_q(A^{\texttt{pStm}})$. Let $\sigma = \sigma_0, \sigma_1 \ldots \sigma_{n-1}$, where $\sigma_i \in \Sigma_{\texttt{pStm}}$, for $i \in [0, n-1]$. The state q is a reachable state in $A^{\texttt{pStm}}$ and, in particular, it is reached by the last partial statement of σ, namely (σ_{n-1}). For our hypothesis, from q it is possible to read $\mathcal{S}(\sigma')$, hence, `Build`(A, q) will be called at line 6 of Alg. 2. Since the lemma is independent from the state, we can apply the same cases showed in the base, starting from q and showing that also the pair (σ', q') will be added to the set I_q, and consequently to $A^{\texttt{pStm}}$. Concluding, since $\sigma \in \mathscr{L}_q(A^{\texttt{pStm}}) \wedge (\sigma' q') \in I_q \Rightarrow \delta = \sigma\sigma' \in \mathscr{L}_{q'}(A^{\texttt{pStm}})$. Since the generalized lemma has been proved for a generic state $q \in Q^A$, Lemma 4.2 also holds when $q \in F^A$.

□

LEMMA A.1. *Let* $\mathsf{r}_1, \mathsf{r}_2 \in RE$. *By construction of* $\wr \cdot \int$ *and* CFG *the following facts hold.*

$$Paths(\texttt{CFG}(\wr \mathsf{r}_1 \| \mathsf{r}_2 \int)) = \left\{ \texttt{true}\, \pi\, \texttt{true} \;\middle|\; \pi \in Paths(\texttt{CFG}(\wr \mathsf{r}_1 \int)) \vee \pi \in Paths(\texttt{CFG}(\wr \mathsf{r}_2 \int)) \right\}$$

$$Paths(\texttt{CFG}(\wr (\mathsf{r}_1)^* \int)) = \left\{ \texttt{true}(\pi\, \texttt{true})^n \;\middle|\; \pi \in Paths(\texttt{CFG}(\wr \mathsf{r}_1 \int)), n \in \mathbb{N} \right\}$$

PROOF. The proof follows by the construction of $\wr \cdot \int$ and CFG and can be easily proved by induction of the structure of the regular expressions. □

PROOF OF LEMMA 4.4. We recall that $\mathcal{S}$ converts a string of strings (in $(\Sigma^*)^*$) in a string of chars (in Σ^*), and *tocode* interprets a string as a executable code, if possible. Let us prove by induction on the structure of r.

- Let $\mathsf{r} = \mathsf{d}$ and $tocode(\mathcal{S}(\mathsf{d})) \in \mu$JS, then $\mathsf{G}_\mathsf{r} = \texttt{CFG}(\wr tocode(\mathcal{S}(\mathsf{d})) \int)$. By Eq. 1 we have that $\forall\, \mathbb{m} \in \mathbb{M}. \exists \Pi \subseteq Paths(\mathsf{G}_\mathsf{r}). [\![tocode(\mathcal{S}(\mathsf{d}))]\!]\, \mathbb{m} \sqsubseteq \bigsqcup_{\pi \in \Pi} [\![\pi]\!]\, \mathbb{m}$.

- Let $\mathsf{r} = \mathsf{d}$ and $\mathit{tocode}(\mathcal{S}(\mathsf{d})) \notin \mu\text{JS}$, then $\mathsf{G_r} = \mathtt{CFG}(\mathbf{skip})$. In this case, $[\![\mathit{tocode}(\mathcal{S}(\mathsf{d}))]\!]\,\mathbb{m} = \mathbb{m}$ since it cannot modify memories not being a legal statement. By construction, $\forall \pi \in \mathit{Paths}(\mathtt{CFG}(\mathbf{skip}))$ we have that, for any $\mathbb{m} \in \mathbb{M}$, $[\![\mathit{tocode}(\mathcal{S}(\delta))]\!] = [\![\pi]\!]\,\mathbb{m} = \mathbb{m}$, hence trivially we have the thesis.

Let us prove now the inductive steps.

- Let $\mathsf{r} = \mathsf{r}_1 || \mathsf{r}_2$. Since a string $\delta \in \mathscr{L}(\mathsf{r}_1||\mathsf{r}_2)$ can be either belong to $\mathscr{L}(\mathsf{r}_1)$ or $\mathscr{L}(\mathsf{r}_2)$, we can split the proof in two cases. We show the proof when $\delta \in \mathscr{L}(\mathsf{r}_1)$, the case $\delta \in \mathscr{L}(\mathsf{r}_2)$ is analogous. Let $\delta \in \mathscr{L}(\mathsf{r}_1)$. Let $\mathsf{G}_{\mathsf{r}_1} \triangleq \mathtt{CFG}(\wr \mathsf{r}_1 \int)$. Then, for inductive hypothesis, $\forall \delta \in \mathscr{L}(\mathsf{r}_1)$. $\forall\, \mathbb{m} \in \mathbb{M}\ \exists \Pi_1 \subseteq \mathit{Paths}(\mathsf{G}_{\mathsf{r}_1})$ s.t. $[\![\mathit{tocode}(\mathcal{S}(\delta))]\!]\,\mathbb{m} \sqsubseteq \bigsqcup_{\pi_1 \in \Pi_1} [\![\pi_1]\!]\,\mathbb{m}$. Let $\mathsf{G_r} \triangleq \mathtt{CFG}(\wr \mathsf{r} \int)$ and $\Pi = \left\{ \mathbf{true}\,\pi_1\,\mathbf{true} \;\middle|\; \pi_1 \in \Pi_1 \right\}$. By Lemma A.1, $\Pi \subseteq \mathit{Paths}(\mathsf{G_r})$. Moreover, since $[\![\mathbf{true}]\!]\,\mathbb{m} = \mathbb{m}$, namely **true** semantics does not alter the input memory, the thesis holds.

$$
\begin{aligned}
&[\![\mathit{tocode}(\mathcal{S}(\delta))]\!]\,\mathbb{m} \sqsubseteq \bigsqcup_{\pi_1 \in \Pi_1} [\![\pi_1]\!]\,\mathbb{m} && \wr \text{Inductive hp.} \int \\
&= \bigsqcup_{\pi_1 \in \Pi_1} [\![\mathbf{true}]\!] \circ [\![\pi_1]\!] \circ [\![\mathbf{true}]\!]\,\mathbb{m} && \wr\, \mathbf{true} \text{ semantics} \int \\
&= \bigsqcup_{\pi_1 \in \Pi_1} [\![\mathbf{true}\,\pi_1\,\mathbf{true}]\!]\,\mathbb{m} = \bigsqcup_{\pi \in \Pi} [\![\pi]\!]\,\mathbb{m} && \wr \text{Compositionality of } [\![\cdot]\!], \text{Def. of } \Pi \int
\end{aligned}
$$

- Let $\mathsf{r} = \mathsf{r}_1\mathsf{r}_2$, $\mathsf{G}_{\mathsf{r}_1} = \mathtt{CFG}(\wr \mathsf{r}_1 \int)$ and $\mathsf{G}_{\mathsf{r}_2} = \mathtt{CFG}(\wr \mathsf{r}_2 \int)$. Then, for inductive hypothesis, $\forall\, \mathbb{m} \in \mathbb{M}\ \forall \delta_1 \in \mathscr{L}(\mathsf{r}_1)$. $\exists \Pi_1 \subseteq \mathit{Paths}(\mathsf{G}_{\mathsf{r}_1})$ s.t. $[\![\mathit{tocode}(\mathcal{S}(\delta_1))]\!]\,\mathbb{m} \sqsubseteq \bigsqcup_{\pi_1 \in \Pi_1} [\![\pi_1]\!]\,\mathbb{m}$ and $\forall \delta_2 \in \mathscr{L}(\mathsf{r}_2)$. $\exists \Pi_2 \subseteq \mathit{Paths}(\mathsf{G}_{\mathsf{r}_2})$ s.t. $[\![\mathit{tocode}(\mathcal{S}(\delta_2))]\!]\,\mathbb{m} \sqsubseteq \bigsqcup_{\pi_2 \in \Pi_2} [\![\pi_2]\!]\,\mathbb{m}$. Let $\mathsf{G_r} = \mathtt{CFG}(\wr \mathsf{r} \int) = \mathtt{CFG}(\wr \mathsf{r}_1\mathsf{r}_2 \int)$ and $\delta \in \mathscr{L}(\mathsf{r}_1\mathsf{r}_2)$. Clearly, $\exists \delta_1 \in \mathscr{L}(\mathsf{r}_1), \delta_2 \in \mathscr{L}(\mathsf{r}_2)$. $\delta_1\delta_2 = \delta$. W.l.o.g., let us suppose that $\delta_1, \delta_2 \in \mu\text{JS}$, since, by definition of $\wr \cdot \int^P$, any non-executable string would be discarded and replaced by **skip**, and the thesis would trivially holds (i.e., $\mathsf{G_r}$ would corresponds to the CFG of **skip** and $[\![\mathit{tocode}(\delta)]\!]\,\mathbb{m} = [\![\mathbf{skip}]\!]\,\mathbb{m}$). Hence, the paths of $\mathsf{G_r}$ are $\Pi = \left\{ \pi_1\pi_2 \;\middle|\; \pi_1 \in \Pi_1, \pi_2 \in \Pi_2 \right\}$.

$$
\begin{aligned}
&[\![\mathit{tocode}(\mathcal{S}(\delta))]\!]\,\mathbb{m} = [\![\mathit{tocode}(\mathcal{S}(\delta_1\delta_2))]\!]\,\mathbb{m} = [\![\mathit{tocode}(\mathcal{S}(\delta_2))]\!] \circ [\![\mathit{tocode}(\mathcal{S}(\delta_1))]\!]\,\mathbb{m} \\
&\sqsubseteq \bigsqcup_{\pi_2 \in \Pi_2} [\![\pi_2]\!]\Big(\bigsqcup_{\pi_1 \in \Pi_1} [\![\pi_1]\!]\,\mathbb{m} \Big) && \wr \text{Inductive hp.} \int \\
&= \bigsqcup_{\pi_1 \in \Pi_1, \pi_2 \in \Pi_2} [\![\pi_2]\!] \circ [\![\pi_1]\!]\,\mathbb{m} && \wr \text{Additivity of } [\![\cdot]\!] \int \\
&= \bigsqcup_{\pi_1 \in \Pi_1, \pi_2 \in \Pi_2} [\![\pi_1\pi_2]\!]\,\mathbb{m} = \bigsqcup_{\pi \in \Pi} [\![\pi]\!]\,\mathbb{m} && \wr \text{Compositionality of } [\![\cdot]\!], \text{Def. of } \Pi \int
\end{aligned}
$$

- Let $\mathsf{r} = (\mathsf{r}_1)^*$. Let $\mathsf{G}_{\mathsf{r}_1} = \mathtt{CFG}(\wr \mathsf{r}_1 \int)$. Then, for inductive hypothesis, $\forall \delta_1 \in \mathscr{L}(\mathsf{r}_1)$. $\forall\, \mathbb{m} \in \mathbb{M}\ \exists \Pi_1 \subseteq \mathit{Paths}(\mathsf{G}_{\mathsf{r}_1})$ s.t. $[\![\mathit{tocode}(\mathcal{S}(\delta_1))]\!]\,\mathbb{m} \sqsubseteq \bigsqcup_{\pi_1 \in \Pi_1} [\![\pi_1]\!]\,\mathbb{m}$ Let $\mathsf{G_r} = \mathtt{CFGGen}((\mathsf{r}_1)^*)$ and $\Pi = \left\{ \mathbf{true}(\pi_1\,\mathbf{true})^n \;\middle|\; \pi_1 \in \Pi_1 \right\}$. By Lemma A.1, $\Pi \subseteq \mathit{Paths}(\mathsf{G}_{(\mathsf{r}_1)^*})$. Let $\delta_1 \in \mathscr{L}(\mathsf{r}_1)$ and $\delta = (\delta_1)^n \in \mathscr{L}((\mathsf{r}_1)^*)$, for some $n \in \mathbb{N}$. Since r is regular expression obtained from a cycle executable automaton, $\delta_1 \in \mu\text{JS}$. Then, $\forall\, \mathbb{m} \in \mathbb{M}$ we have the thesis.

$$
\begin{aligned}
&[\![\mathit{tocode}(\mathcal{S}(\delta))]\!]\,\mathbb{m} = [\![\mathit{tocode}(\mathcal{S}((\delta_1)^n))]\!]\,\mathbb{m} = ([\![\mathit{tocode}(\mathcal{S}(\delta_1))]\!])^n\,\mathbb{m} && \wr \delta_1 \in \mu\text{JS} \int \\
&\sqsubseteq \bigsqcup_{\pi_1 \in \Pi_1} ([\![\pi_1]\!]\,\mathbb{m})^n && \wr \text{Inductive hp.} \int \\
&= [\![\mathbf{true}]\!] \circ \bigsqcup_{\pi_1 \in \Pi_1} ([\![\mathbf{true}]\!] \circ [\![\pi_1]\!]\,\mathbb{m})^n && \wr\, \mathbf{true} \text{ semantics} \int \\
&= [\![\mathbf{true}]\!] \circ \bigsqcup_{\pi_1 \in \Pi_1} ([\![\pi_1\,\mathbf{true}]\!]\,\mathbb{m})^n = \bigsqcup_{\pi_\epsilon \Pi} [\![\pi]\!]^{\#}\,\mathbb{m} && \wr \text{Compositionality of } [\![\cdot]\!], \text{Def. of } \Pi \int
\end{aligned}
$$

□