Abstract Code Injection
A Semantic Approach Based on Abstract Non-Interference

Samuele Buro and Isabella Mastroeni

Department of Computer Science, University of Verona
Strada le Grazie 15, 37134 Verona, Italy
{samuele.buro,isabella.mastroeni}@univr.it

Abstract. Code injection attacks have been the most critical security risks for almost a decade. These attacks are due to an *interference* between an *untrusted input* (potentially controlled by an attacker) and the execution of a *string-to-code* statement, interpreting as code its parameter. In this paper, we provide a semantic-based model for code injection parametric on what the programmer considers *safe behaviors*. In particular, we provide a *general* (abstract) non-interference-based framework for *abstract code injection* policies, i.e., policies characterizing safety against code injection w.r.t. a given specification of safe behaviors. We expect the new semantic perspective on code injection to provide a deeper knowledge on the nature itself of this security threat. Moreover, we devise a mechanism for enforcing (abstract) code injection policies, *soundly* detecting attacks, i.e., avoiding false negatives.

1 Introduction

Security is an enabling technology, hence security means power. The correct functionality and coordination of large scale organizations, e-government, web services, in general, relies on confidentiality and integrity of data exchanged between different agents, and on the proper functioning of the applications. These features, almost unavoidable, become real opportunities for the attackers seeking to disclose and/or corrupt valuable information or, more widely, to break security.

According to OWASP (Open Web Application Security Project) [1], the most critical security risks have been application level injections attacks for almost a decade [21,22,23]. The reason of their success and their spread is twofold: An *easy exploitability* of vulnerabilities and a *severe impact* of attacks. In other words, code injection bugs allow attackers to cause extensive damage for minimum effort. Despite this, organizations often underestimate their consequences, and the inevitable result has been a recent history full of this kind of attacks [29].

Several approaches [3,10,11,18,19,24,28,30,31], have been studied for *preventing* code injection, but only few focus on the harder problem of *defining* it [3,24,28]. Indeed, the intuition of what can be classified as an injection attack is quite straightforward, and it is clearly provided in the following informal definition [23]:

"Injection occurs when user-supplied data is sent to an interpreter as part of a command or query. Attackers trick the interpreter into executing unintended

```
<?php

$id = $argv[0];
$query  = "SELECT * FROM users WHERE id = $id;";
$result = pg_query($conn, $query);

?>
```

```
<html><body>
<%
   String user =
      request.getParameter("user");
%>
<h1>Welcome <%= user %></h1>
</body></html>
```

Fig. 1: Example of SQLi in a PHP program and of XSS in a JSP page.

> commands via supplying specially crafted data. Injection flaws allow attackers to create, read, update, or delete any arbitrary data available [. . .].”

Unfortunately, this intuitive definition does not help in formalizing a general definition of code injection since there is a clear problem in formalizing the concept of *unintended commands*. Moreover, this notion may depend on several factors, e.g., the kind of application, the environment of execution, etc.

The essence of code injection. Code injection is a wide category of attacks where an attacker exploits the presence of an *untrusted input* (i.e., an input whose source is potentially untrusted) for injecting code (*unintended commands*) that will affect the execution of a *string-to-code* statement (which interprets as code its parameter), altering the output behavior of the application.

The main types of attacks in this category are undoubtedly SQL injection (SQLi) and cross-site scripting (XSS) attacks. In SQLi the attacker attempts to execute an arbitrary query on a database server. An example is given on the left in Fig. 1 where the attacker is able to extract the whole content of the users table by injecting the value `"3 OR 1 = 1"` in the untrusted input `argv[0]`, making the `query` condition a tautology. In XSS the attacker attempts to execute a client-side code (e.g., JavaScript) in the user’s browser. In the program on the right in Fig. 1, an attacker can execute his own JavaScript code in the victim’s browser by injecting, for example, the string `"<script>alert('message')</script>"`. Other kinds of code injection attacks such as command injection, `eval`-injection, XPath injection, remote file injection, etc., also play a central role in this category.

Despite the multi-faceted nature of code injection, all these attacks present a key common feature: *A code executed by an interpreter which is* dependent on *the value of an untrusted input, that alters the intended semantics of the application, making the execution unsafe.*

Up to date solutions for facing code injection. The prevention techniques against code injection are a well-studied topic of applications security, and they can be classified into two categories: The techniques that follow an industrial/technical approach and those based on formal methods.

In the former case, applications are made secure by validating inputs (escaping, whitelists, blacklists, etc., of values), by parametrizing queries (i.e., by separating the parameters binding from the code compilation — mainly used in the SQLi context) and/or by using other ad hoc mechanisms. Even though they provide some degree of security and robustness to programs, they suffer from well-known flaws [2,10,13,18,30]. Moreover, they do not aid the programmers through the process of securing applications.

In the latter case, there are several formal approaches claimed to be *sound* and/or *complete* w.r.t. a given notion of code injection [28,31,10,24]. The majority of them are mainly focused on the SQLi attacks than on the broader problem of code injection, and they rely on dynamic taint analysis algorithms [11,19,24,31] or parsing trees [4,28] in order to detect alterations of the syntactic queries structure on the basis of fixed policies. To the best of our knowledge, the works that mainly rely on the problem of *defining* code injection are [3,24,28]. All of them provide a *syntactic-based* notion of code injection and the two most related to our approach are [3,24]:

- In [3], the core idea is to dynamically mine the programmer-intended query structure on any input, and to detect attacks by comparing them against the intended query structure;
- In [24], an application's output is considered a code injection attack if there exists at least one tainted symbol which is *code*, i.e., it is not a fully evaluated value.

It is worth noting that these definitions are indeed specific instances of the informal definition given in the introduction. In particular, both fix a precise notion of what a programmer could consider as *unintended commands*. This loss of generality is clearly useful in practice, since it provides a decidable and easy way to detect potential code injection attacks, but it may reduce flexibility, since the programmer might need to weaken or strengthen the fixed notion, depending of the environment of execution of developed applications.

Our solution: A semantic-based approach. In this paper, we propose to shift these syntactic notions towards a semantic model of code injection — as suggested in [24] — in order to broaden the generality of the definition.

The key point of the whole approach we propose is based on a simple observation: Each time an expression e is *executed* in a string-to-code statement (e.g., in a query execution, in an `eval` statement, ...), there is a set of states[1] such that the execution of e in one of these states leads the program to an unintended/unsafe state. Since we are focusing on code injection, we can restrict this set only to those states *depending* on at least one injected (untrusted) value. In other words, we have injection whenever there is a program statement whose parameters *depend on* an untrusted input and whose execution causes an *unsafe output behavior*, namely the attacker can lead the program execution to show unsafe behaviors. For instance, in the code on the left of Fig. 1, the query execution statement `pg_query` depends on the untrusted input `argv[0]`.

[1] Intuitively, think of a state as all the information concerning the program execution at each step of computation.

This kind of data dependency is precisely what is called *interference* in language-based security [9,26]. This means that, safety against code injection can be seen as a non-interference policy. Being more precise, it is clear that we do not care to model any possible interference, since any *dynamic* code is expected to depend on the input in some way. There is a potential security breach only when the dependency causes a variation between what is considered safe and what is considered unsafe. In this sense, the right non-interference framework to consider is *abstract non-interference* [8], where the interference between *properties* of inputs and *properties* of outputs is studied. Hence, we define *abstract code injection policies* parametric on what the programmer considers safe output behaviors.

It is clear that, if we could provide a universal characterization of what is a *safe output behavior* (holding for all programs, for all execution environments), we could design a tool enforcing (abstract) code injection policies for any program.

Unfortunately, in real settings, different programs, or even the same program in different execution contexts, may require different instances of the policy. Consider a music streaming web application P_{music} with premium and free users. The first ones have access to both copyright and copyright-free music, while the second ones can only listen to copyright-free songs. In order to encourage free users to buy premium subscriptions, the programmers allow them to add a copyrighted song in their music library once per month, chosen from a list of top five hits. Suppose the user chooses the first song and its code number (e.g., `83`) is submitted to the web application as a GET parameter: `https://webappmusic.com/load_library.php?choice=83`. To load user's library, the web application executes the following query where the variable `$free_codes_list` contains the codes of all the copyright-free songs and the variable `$user_choice` contains the code `83`[2]:

$$\begin{array}{c}\texttt{SELECT * FROM songs WHERE code IN (\$free\_codes\_list, \$user\_choice)}\\ \downarrow \\ \texttt{SELECT * FROM songs WHERE code IN (5, 3, 2, 54, 32, 21, 12, ..., }\underline{\texttt{83}}\texttt{)}\end{array}$$

If the GET parameter `choice` is not validated, an evil user can inject an arbitrarily long list of values, loading more than one song in his/her library, for instance

$$\texttt{SELECT * FROM songs WHERE code IN (5, 3, 2, 54, 32, 21, 12, ..., }\underline{\texttt{83, 43, 23, ...}}\texttt{)}$$

Consider now a web application P_{doc} allowing users to download documents from a list of `pdf` files. An user provides the documents' codes he/she wishes to download (e.g., `2, 23, 6`) and an HTTP request is sent to the web application: `https://webappdoc.com/download.php?doc[]=2&doc[]=23&doc=6`. Suppose the user's choice is stored in the PHP variable `$doc_list` and the following query is executed:

$$\begin{array}{c}\texttt{SELECT * FROM docs WHERE code IN (\$doc\_list)}\\ \downarrow \\ \texttt{SELECT * FROM docs WHERE code IN (}\underline{\texttt{2, 23, 6}}\texttt{)}\end{array}$$

[2] The highlighted code is the injected one.

In this case, a list of values has been injected but, contrarily to the P_{music} scenario, it is not to be considered as an attack, since the programmer's intention and the context are different. It follows that every model fixing a notion of *unintended commands* will provide a wrong answer to, at least, one of the two examples. Even worse, let us change the first example by supposing that the programmers decide to allow the user to choose two songs: A list of two songs now has not to be considered an attack. These trivial examples show how, even in the same context, the programmer's intention, and therefore what is unintended, may vary.

It is worth noting that a common feature in these examples is that, for controlling code injection we have to partition inputs into two subsets: The set of inputs producing safe output behaviors after the query execution, and all the others, generating unsafe behaviors. With these considerations in mind, the model we propose is based on the following key points:

1. Abstract code injection policies can be defined in terms of an output characterization of safe output behaviors, potentially determined by the programmer;
2. If the program does not satisfy this policy, it means that there are values of untrusted inputs able to change the output (observable) behavior of the program, making it unsafe. We call *safe inputs* those always leading to safe output behaviors;
3. The abstract non-interference framework [8] allows us to characterize the partition of inputs leading to different (safe/unsafe) output behaviors. This suggests us what should be verified on the input of the application.

At this point, in order to control code injection vulnerabilities we propose to go through two phases: First, we characterize the abstract code injection policy to enforce, for instance by asking the programmer to specify safe inputs and/or safe output behaviors; Second, we enforce the chosen policy. The latter phase could be tackled both statically, by manually patching the program (but in this case we lose flexibility), or by monitoring the program, i.e., by dynamically checking whether the executed inputs are safe, w.r.t. some decidable characterization.

Hence, we propose a static analysis for aiding the programmer to understand when a safe input specification is necessary, and consequently asking the programmer to annotate the program with information characterizing the abstract injection policy to enforce. Then we propose the design of a dynamic analysis, i.e., a monitor checking whether the execution violate the abstract injection policy.

2 Background

The core language WhileFun. In order to show our approach, we define the core language WhileFun that encloses all the important features from the code injection point of view. WhileFun is dynamically typed, based on a classic While language augmented with functions. A valid WhileFun program (denoted by $P \in$ WhileFun) consists in a `main` function (the entry point, non-callable by the code) and eventual user-defined functions. We assume that only a subset of the parameters of the `main` function may be *untrusted inputs*. Furthermore, we introduce the syntactic category *str2code* of string-to-code statements:

```
str2code ::= exec(exp)    SQL query execution      (SQL injection)
           | eval(exp)    Code execution           (eval-injection)
           | system(exp)  Command/Shell execution  (Command injection)
           | show(exp)    Webpage displaying       (XSS)
```

All these commands send the evaluated expression *exp* (a string of code) to the corresponding interpreter. For the sake of simplicity, we assume that *str2code* commands are only allowed in the `main` function. The language syntax and the semantics of the other commands are standard.

Program semantics. $\mathcal{Vars}$ denotes the set of program and environment variables[3], and $\mathcal{Val}$ the set of values. $\mathcal{L}$ denotes the set of *line numbers* (program points). Let $l \in \mathcal{L}$, and $Stm(l)$ be the statement at program line l. For a given program P, we denote by $\mathcal{L}_P \subseteq \mathcal{L}$ the set of all and only the line numbers corresponding to statements of the program P, i.e., $\mathcal{L}_P = \{ l \in \mathcal{L} \mid Stm(l) \in P \}$.

A *program state* $\sigma \in \mathcal{S}$ is a pair $\langle n^k, \mu \rangle$ where n is the executed program point, k is the number of times the statement $Stm(n)$ has been reached so far (in the following we will call n^k *execution point*), μ is the memory [17]. A *memory* $\mu \in \mathcal{Mem}$ is a map $\mu : \mathcal{Vars} \to \mathcal{Val}$ mapping variables to values such that $\mu(x)$ is the value of x in μ, while $\mu[x \leftarrow v]$ is the memory μ' such that $\forall y \neq x. \mu'(y) = \mu(y)$, while $\mu(x) = v$. For simplicity, we denote by $\mathcal{Val}^x$ the set of values over which x can range, i.e., the domain of x. Furthermore, we define the equivalence relation $=_x$ between two memories μ and μ': $\mu =_x \mu' \iff \forall y \neq x.\ \mu(y) = \mu'(y)$.

A *state trajectory* $\tau \in \mathcal{T} = \mathcal{S}^* \cup \mathcal{S}^\omega$ is a sequence of program states through which a program goes during the execution. Any initial state has $n^k = 1^1$, i.e., the set of initial states is $\mathcal{S}_\iota = \{ \langle 1^1, \mu \rangle \mid \mu \in \mathcal{Mem} \}$. The state trajectory obtained by executing program P from the input memory μ is denoted by $\langle\!| P |\!\rangle(\langle 1^1, \mu \rangle)$ and $\langle\!| P |\!\rangle^{n^k}(\langle 1^1, \mu \rangle)$ is the prefix of $\langle\!| P |\!\rangle(\langle 1^1, \mu \rangle)$ whose last state has execution point n^k. The denotational semantics of $P \in$ WhileFun is the function $[\![P]\!] : \mathcal{S} \to \mathcal{S}$ providing the I/O characterization of program semantics. Let $\langle 1^1, \mu_\iota \rangle \in \mathcal{S}_\iota$, the denotational semantics is defined as $[\![P]\!](\langle 1^1, \mu_\iota \rangle) = \sigma_\dashv$ where $\sigma_\dashv$ is the last state of $\langle\!| P |\!\rangle(\langle 1^1, \mu_\iota \rangle)$ if it is finite, $\perp$ otherwise [6]. We similarly define $[\![P]\!]^{n^k}(\langle 1^1, \mu_\iota \rangle)$, the denotational semantics w.r.t. to the *execution point* n^k.

Static Single Assignment (SSA) SSA [7] is a well known code representation where the def-use chains are made explicit. This is an intermediate non-executable representation of code, used by compilers for simplifying some static analyses. In the SSA form, each assignment generates a new unique name (usually denoted by a numerical subscript) for the defined variable, and all the uses reached by that definition are renamed. An example is shown in Fig. 2a where the program on the left is rewritten in the one on the right. If different definitions reach the

[3] Without loss of generality, we assume that the state of the *str2code* interpreter (for instance, the state of the database when the interpreter is the database server) is modeled in the set of variables $\mathcal{Vars}$. This means that the memory μ contains all the observable information concerning both the program and environment.

```
V := 4          V₁ := 4
Z := V + 5      Z₁ := V₁ + 5
V := 6          V₂ := 6
W := V + 7      W₁ := V₂ + 7
```

(a) Linear SSA transformation.

```
if (P)                  if (P)
  then { V := 4 }         then { V₁ := 4 }
  else { V := 6 }         else { V₂ := 6 }
                        V₃ := ϕ(V₁, V₂)
```

(b) SSA transformation with ϕ-function.

Fig. 2: Examples of SSA program representations [7].

same use of a given identifier, a special form of assignment, called *ϕ-function*, is added: This is a special assignment identifying the join of several definitions of the same identifier (an example is given in Fig. 2b). The presence of these ϕ-functions makes the code not-executable but there exist standard techniques for reconstructing executable programs from the SSA form: By replacing the ϕ-functions with assignment operations, and by dropping subscripts [7].

Reaching definitions analysis (RD). Reaching definitions analysis (RD for short), determines the definitions potentially reaching each use of an identifier. In a control flow graph (CFG), a definition reaches a node if there is a path from the definition to the node, along which the defined variable is never redefined. On the SSA form this analysis becomes trivial since the reaching definition is precisely the unique definition of the used identifier (see [20] for details).

3 Defining Abstract Code Injection

In this section, we define the notion of *abstract code injection* policy, which consists in a code injection policy *parametric on* the programmer characterization of safe/unsafe output behaviors. More specifically, a code injection vulnerability is a potential *interference* between an untrusted input and the execution of a string-to-code statement. We say that a program does not suffer of a code injection vulnerability if it *enforces a code injection policy*, meaning that any code injection vulnerability in the program is avoided.

First of all, let us define formally code injection policies in terms of non-interference. In particular, we define a notion of non-interference between an input and a program point, e.g., the program point of the string-to-code statement. We recall that $[\![P]\!]^{n^k}$ (see Sect. 2) computes the state at the execution point n^k.

Definition 1 **($\mathrm{NI}^x_P(n)$).** *Let $P \in$ WhileFun be a program, x be an input of P and $n \in \mathcal{L}_P$. We say that x is* non-interfering *at the program point n of P iff*

$$\forall k \in \mathbb{N}\,.\ \mathrm{NI}^x_P(n,k)$$

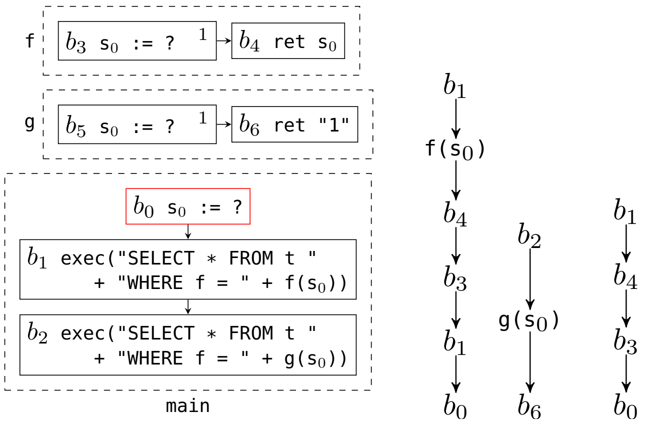
where, for any $k \in \mathbb{N}$, $\mathrm{NI}^x_P(n,k)$ (x non-interfering *at the execution point n^k in P) holds iff*

$$\mu_0 =_x \mu_0' \implies [\![P]\!]^{n^k}(\langle 1^1, \mu_0\rangle) = [\![P]\!]^{n^k}(\langle 1^1, \mu_0'\rangle)$$

```
f(s) { ³ret s   };
g(s) { ⁴ret "1" };

main(s) {
¹exec("SELECT * FROM t "
  + "WHERE f = " + f(s));
²exec("SELECT * FROM t "
  + "WHERE f = " + g(s))
}
```

(a) Program P source code.



(b) CFG of the SSA form of P. (c) Trees generated by *Paths*. (d) Cleaned trees.

Fig. 3: Steps of static analysis algorithms.

Intuitively, this notion states that, during the execution of P, whenever the execution reaches the program point n, even if we change the initial value of x, the observable behavior of P does not change. It is self-evident that an attacker cannot perform a successfully injection attack on x if the above definition holds for all the program points where a string-to-code statement is executed.

For instance, consider the string-to-code statement ¹`exec("SELECT * FROM t WHERE f = " + f(s))` in Fig. 3a. There exist two values $v_1 = 1, v_2 = 2$ generating two different queries (observable behaviors) after its execution. On the other hand, if we consider the statement ²`exec("SELECT * FROM t WHERE f = " + g(s))`, then for each $v_1, v_2 \in \mathcal{Val}^{\mathtt{s}}$ we have that $[\![P]\!]^{2^1}(\langle 1^1, \{\, \mathtt{s} \to v_1 \,\}\rangle) = [\![P]\!]^{2^1}(\langle 1^1, \{\, \mathtt{s} \to v_2 \,\}\rangle)$ since `g(s)` is a constant function.

Exactly as it happens in language-based security, this notion of non-interference (and therefore of code injection policy) is in general too strong, since it does not allow us to really distinguish between safe and potentially unsafe code. In particular, this definition says that the only safe code is the one not depending on untrusted inputs, which is in general not acceptable: String-to-code statements, such as query executions, code evaluations, etc., *have* to be dependent on the user's input. For this reason, we need to formalize code injection parametrically on what the programmer considers a *safe (output) behavior* and/or which are the programmer *(expected) safe inputs*, leading only to safe outputs.

Formally, let $\mathcal{O} \subseteq \mathcal{Mem}$ be the set of all the output states considered safe by the programmer after the execution of an string-to-code statement ($\mathcal{O}$ is the set of safe output behaviors). We can define the characteristic map of $\mathcal{O}$ as

$$\rho_{\mathcal{O}}(\langle n^k, \mu\rangle) = \begin{cases} \mathit{true} & \text{if } \mu \in \mathcal{O} \\ \mathit{false} & \text{otherwise} \end{cases}$$

At this point we can weaken Def. 1 defining abstract code injection policies, parametric on the programmers characterization of safe outputs $\mathcal{O}$ as done for abstract non-interference [8,14,15].

Definition 2 **($\mathrm{ANI}^{x}_{P}(\mathcal{O}, n)$).** *Let $P \in$ WHILEFUN be a program, x be an input of P, $n \in \mathcal{L}_P$ and $\mathcal{O}$ be the set of the safe output behaviors. We say that x is* non-interfering *w.r.t. $\mathcal{O}$ at the program point n in P iff*

$$\forall k \in \mathbb{N} \,.\; \mathrm{ANI}^{x}_{P}(\mathcal{O}, n, k)$$

where, for any $k \in \mathbb{N}$, $\mathrm{ANI}^{x}_{P}(\mathcal{O}, n, k)$ (x is (abstract) non-interfering *w.r.t. $\mathcal{O}$ with the execution point n^k in P) iff*

$$\mu_0 =_x \mu_0' \implies \rho_{\mathcal{O}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0 \rangle)) = \rho_{\mathcal{O}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0' \rangle))$$

We say that a program enforces an *abstract code injection* policy, w.r.t. a safe output behaviors characterization $\mathcal{O}$, if does not exist any untrusted inputs x and string-to-code statements in P (at the program line n) such that $\neg\, \mathrm{ANI}^{x}_{P}(\mathcal{O}, n)$.

The abstract non-interference framework [8,14,15] allows us to move further and to characterize an enforcing strategy when an abstract code injection policy is not satisfied. It should be clear that also $\mathrm{ANI}^{x}_{P}(\mathcal{O}, n)$ is a too strong property, in the sense that most "raw" programs cannot satisfy it, unless they show, independently from the input, always the same kind of behavior (safe or unsafe).[4] For all the other programs, where attackers have the possibility of exploiting an untrusted input for leading to unsafe output behaviors, namely those *vulnerable* to code injection, the abstract non-interference framework allows us to characterize which variation of inputs causes the safe/unsafe variation of output behaviors. In other words, in this framework it is possible to determine the input binary partition for every input x (defined by the characterization function $\phi^x : \mathcal{V}al^x \to \{\mathit{true}, \mathit{false}\}$) making the following equation to hold for each $v_1, v_2 \in \mathcal{V}al^x$ [8,16]:

$$\phi^x(v_1) = \phi^x(v_2) \implies \rho_{\mathcal{O}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_1] \rangle)) = \rho_{\mathcal{O}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_2] \rangle))$$

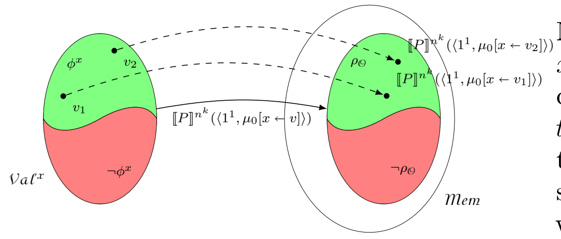

ϕ^x
v_2
v_1
$\neg\phi^x$
$\mathcal{V}al^x$
$\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v] \rangle)$
$\rho_{\mathcal{O}}$
$\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_2] \rangle)$
$\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_1] \rangle)$
$\neg\rho_{\mathcal{O}}$
$\mathcal{M}em$

Fig. 4: The bipartions induced by $\mathcal{O}$.

Namely, for each pair of values for x, both in the same equivalence class of ϕ^x ($\phi^x(v_1) = \phi^x(v_2) = \mathit{true}$ or $\phi^x(v_1) = \phi^x(v_2) = \mathit{false}$) the output behavior is always respectively safe or unsafe. In Fig. 4 we depict the situation. All and only the values in $\mathcal{V}al^x$, satisfying ϕ^x leads the execution of P in n^k to satisfy $\rho_{\mathcal{O}}$.

We can formally characterize the partition ϕ^x *enforcing* an abstract code injection policy $\mathrm{ANI}^{x}_{P}(\mathcal{O}, n)$ (simply denoted $\phi^x_{\mathcal{O}}$), as follows [8]:

$$\forall v \in \mathcal{V}al^x \,.\; (\phi^x_{\mathcal{O}}(v) \iff \rho_{\mathcal{O}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v] \rangle))) \qquad (1)$$

[4] Note that, programs showing always unsafe behaviors are of no interest since they are unsafe by nature, the attacker cannot force unsafety.

It is worth noting that, the set $\mathcal{I}^x_{\mathcal{O}} = \left\{\ v \in \mathcal{V}al^x \,\middle|\, \phi^x_{\mathcal{O}}(v) = \mathit{true}\ \right\}$ is precisely the *language of safe inputs w.r.t.* $\mathcal{O}$, i.e., those inputs leading to safe outputs.

Note that, we call $\phi^x_{\mathcal{O}}$ an *enforcing strategy*, since it characterizes what we should check in order to avoid code injection w.r.t. $\mathcal{O}$. Once we know which are the safe inputs, we could check whether the received inputs, during computation, are safe. The possibilities are two: The programmer could patch the code implementing all the checks, but this reduces the approach flexibility (if $\mathcal{O}$ changes then the code has to be partially rewritten); The programmer could augment the code with input annotations that can be checked dynamically. In the following section, we propose a monitor-based approach. The choice is driven by the idea of having a flexible enforcing technique, allowing the programmer to change $\mathcal{O}$ without changing the code, but only the input annotations. The approach we propose, allows us a further degree of flexibility, allowing us to fix input annotations depending also on the dynamic execution path.

4 Enforcing Abstract Code Injection

In this section, we propose a technique for enforcing code injection policies w.r.t. the programmer characterization of safe output behaviors, namely able to recognize and stop executions potentially under a code injection attack.

In the previous section, we showed that starting from a characterization of safe output behaviors $\mathcal{O}$, we can characterize the language of safe inputs w.r.t. $\mathcal{O}$, i.e., $\mathcal{I}^x_{\mathcal{O}}$ for each input x. Unfortunately, even if $\mathcal{O}$ is decidable, the definition of $\mathcal{I}^x_{\mathcal{O}}$ does not guarantee in general its decidability, hence we propose a technique where the programmer provides a decidable language $\mathcal{I}^x$ of *acceptable* inputs *consistent* with $\mathcal{O}$, namely such that it satisfies the following inclusion

$$\forall v \in \mathcal{V}al^x\,.\,(v \in \mathcal{I}^x \ \Rightarrow\ \rho_{\mathcal{O}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v]\rangle)))$$

meaning that $\mathcal{I}^x \subseteq \mathcal{I}^x_{\mathcal{O}}$. This inclusion guarantees soundness, in the sense that it avoids false negatives, while it can admit false positives/alarms, since there are safe inputs that are not in the acceptable input language.

In general, it should be clear, that both soundness and completeness of the model w.r.t. real potential attack situations depend on the choice of $\mathcal{O}$ and of $\mathcal{I}^x$. An over-approximation of $\mathcal{I}^x$ or $\mathcal{O}$ (meaning that there are inputs or behaviors erroneously labeled as safe) may lead to false negatives, missing some attack situations and therefore losing soundness w.r.t. real potential attacks. An under-approximation of $\mathcal{I}^x$ or $\mathcal{O}$ (meaning that there are safe inputs or behaviors labeled as unsafe) may allow false positives/alarms, hence losing precision/completeness of the approach w.r.t. real potential attacks. In the following, we will always talk of soundness and completeness of the enforcing technique w.r.t. the chosen model, i.e., w.r.t. the choice of all $\mathcal{I}^x$ and/or $\mathcal{O}$.

4.1 A contract-based approach for enforcing abstract code injection

The approach we propose is based on the idea of asking the programmer the language $\mathcal{I}^x$ of acceptable input values for each untrusted input. In order to avoid useless contracts for untrusted inputs not leading to the execution of a string-to-code statement, we aid the programmer providing him/her both the inputs and the paths potentially vulnerable to code injection, and therefore requiring him/her to fix a corresponding language of safe inputs $\mathcal{I}^x$, that we call *contract*. Then we propose to annotate the code with these contracts in order to dynamically monitor the program execution for checking contracts only when necessary, namely when a string-to-code statement, depending on an untrusted input, is executed. The approach we propose is composed by three phases:

- *Static analysis:* The code is statically analyzed in order to extract the vulnerable paths, where some untrusted inputs interfere with the execution of a string-to-code statement, and therefore where the programmer should establish some input restrictions.
- *Contracts request:* Then we ask the programmer the contracts for all the *vulnerable paths*, i.e., those paths where an untrusted inputs reaches a string-to-code statements. This phase could be made automatic by providing a general/unique definition of restrictions, independently from the particular untrusted input and/or executed path. The result of this phase is an *annotated program* with contracts.
- *Monitor:* Finally, the monitor is able to kill all the executions of an annotated program, when a vulnerable path is executed and the involved untrusted input violates its contract. In other words, the result of this phase is a *monitored program*, namely the monitor specialized on the annotated program [12].

Basic notations. In literature, there are many variants of the CFG construction [5]: We choose to build a *single block intraprocedural CFG*, in which we consider the parameters of the `main` function to be potentially *untrusted inputs* (depicted in **red** in Fig. 3b), while the parameters of the other functions are considered *formal parameters* (depicted with a numerical superscript in Fig. 3b). We define the notion of *sub-path* of a path $b_0 \dots b_m$ in a CFG as a sequence of blocks $p_0 \dots p_n$ such that $p_0 = b_0$, $p_n = b_m$, and for each $0 < i < n$ if $p_{i-1} = b_k$ then $\exists b_j, j > k$ such that $p_i = b_j$, intuitively, it is a path of the CFG where some intermediate blocks are missing. Given a CFG C, if there exists a path p in it such that p' is a sub-path of p, then we say that p' is sub-path of C.

In addition, we define the following domains: $\mathcal{Blocks}$ is the set of blocks in the CFG, $\mathcal{FunCalls}$ is the set of all function calls (including arguments) in the SSA form of the program, $\mathcal{Fun}$ is the set of the defined functions, $\mathcal{Args}$ is the set of arguments of the function calls, and here $\mathcal{Vars}$ is the set of program variables in the SSA form. We also use the following well known functions: $\textsc{Use}$ to compute the variables used and the function calls performed in a statement or, by extension, in a block, $\textsc{Ret}$ to compute the set of the returning points of a given function, and $\textsc{Rd}$ to compute the RD analysis. We call *ground block* any block b such that $\textsc{Use}(b) = \varnothing$.

$\mathcal{Trees}$ is the set of trees whose nodes are either blocks, or calls, or $\perp_{k \in \mathbb{N}}$ values. Let n be a tree node and $\mathbf{T} \subseteq \mathcal{Trees}$, the tree constructor is $\mathit{Tree}(n, \mathbf{T}) = \langle n : \mathbf{T} \rangle$ which, starting from a set of several trees $\mathbf{T}$, builds one new tree with root n and sub-trees those in $\mathbf{T}$, i.e., it adds an edge (n, m) in $\langle n : \mathbf{T}\rangle$ for each node m root of a tree in $\mathbf{T}$. $\langle n\rangle \equiv \langle n : \varnothing\rangle$. On a tree T, we define the function $\textsc{Branches}(T)$ that returns the set of all the paths p, from the root to a leaf, in T, and the function $\textsc{Reverse}(p)$ that changes the direction of edges in the path p. We abuse notation by calling $\textsc{Reverse}$ also its additive lift to sets of paths.

4.2 Static phase and contract request

The purpose of the static phase is to detect *where* the information flows within a program under analysis. In particular, we are interested in all the vulnerable paths, i.e., those starting from an untrusted input and affecting a string-to-code statement. We explain our approach and algorithms also by using as running example the code in Fig. 3a.

CFG and SSA construction. Let P be the program under analysis. The first two steps of the analysis consist in the construction of the program representation that will be used for performing the analysis. First, we build the control flow graph for each procedure f declared in the program ($\textsc{Cfg}(f)$) obtaining the set $\mathbb{C} = \{\, \textsc{Cfg}(f) \mid f \text{ procedure in } P\,\}$. After that, in order to improve the analysis (and, in particular, RD), we consider the SSA representation of each CFG (see Sect. 2 for details), where each variable is defined only once. Let $\textsc{Ssa}$ be the function that computes the SSA form of a given CFG. We define the set $\mathbb{C}^{ssa} = \{\, \textsc{Ssa}(C) \mid C \in \mathbb{C}\,\}$ representing the SSA form of the program P (for an example, see Fig. 3b).

Trees construction. The trees construction is the core step of the static analysis: For each block b containing a string-to-code statement, the function *Paths* (Fig. 5) builds, backwards, the trees of potential execution paths, looking for the vulnerable ones.

Paths is a function with two parameters: The first one is either a block $b \in \mathscr{Blocks}$, or a procedure call $f(\mathbf{a}) \in \mathscr{FunCalls}$, or a special value $\bot_k$ ($k \in \mathbb{N}$); The second parameter is a *history* of function calls $\mathbf{c} = [c_1, \ldots, c_k]$ (ε denotes the empty history). A *function call* is a triple $(b, f, \mathbf{a})$ consisting in the calling block b, in the called function f, and in its sequence of actual parameters $\mathbf{a} = [a_1, \ldots, a_m]$. Given a block $b \in \mathscr{Blocks}$ containing a string-to-code statement and an initial empty sequence of calls $\mathbf{c} = \varepsilon$, the function *Paths* tracks backward the potential dependencies in order to identify which untrusted inputs may affect the string-to-code statement in b. These chains of dependencies form a tree having b as root and as leaves either a ground block, or bottom, or a block containing the interfering untrusted input (the latter case identifies vulnerable paths).

In order to formally define the function *Paths*, we have first to define the auxiliary function *Rec*, used for determining the arguments of recursive calls, i.e., whose aim is that of computing the parameters of the recursive step of *Paths*:

$$\begin{aligned}\mathit{Rec}\colon (&\mathscr{Blocks} \times (\mathscr{Vars} \cup \mathscr{FunCalls}) \times (\mathscr{Blocks} \times \mathscr{Fun} \times \mathscr{Args})^*) \to \\ &(\mathscr{Blocks} \cup \mathscr{FunCalls} \cup \{\bot_k\}_{k\in\mathbb{N}}) \times (\mathscr{Blocks} \times \mathscr{Fun} \times \mathscr{Args})^*\end{aligned}$$

$$Paths\colon (\mathcal{Blocks} \cup \mathcal{FunCalls} \cup \{\bot_k\}_{k\in\mathbb{N}}) \times (\mathcal{Blocks} \times \mathcal{Fun} \times \mathcal{Args})^* \to \mathcal{Trees}$$

$$Paths(arg, \mathbf{c}) = \begin{cases} Block(arg, \mathbf{c}) & \text{if } arg \in \mathcal{Blocks} \\ Call(arg, \mathbf{c}) & \text{if } arg \in \mathcal{FunCalls} \\ Bot(\bot_k, \mathbf{c}) & \text{otherwise} \end{cases}$$

where $\mathbf{c} = [c_1, \ldots, c_m]$ such that $\forall 1 \leq i \leq m \,.\, c_i = (b_i, f_i, \mathbf{a}^i)$ and $\mathbf{a}^i = [a^i_1, \ldots, a^i_{n_i}]$

$$Block\colon \mathcal{Blocks} \times (\mathcal{Blocks} \times \mathcal{Fun} \times \mathcal{Args})^* \to \mathcal{Trees}$$
$$Call\colon \mathcal{FunCalls} \times (\mathcal{Blocks} \times \mathcal{Fun} \times \mathcal{Args})^* \to \mathcal{Trees}$$
$$Bot\colon \{\bot_k\}_{k\in\mathbb{N}} \times (\mathcal{Blocks} \times \mathcal{Fun} \times \mathcal{Args})^* \to \mathcal{Trees}$$

$$\begin{cases}
(1) & Block(b, \mathbf{c}) & = Tree(b, \{\, Paths(Rec(b, u, \mathbf{c})) \mid u \in \textsc{Use}(b) \,\}) \quad \text{if } \textsc{Use}(b) \neq \varnothing \\
(2) & Block(b, \mathbf{c}) & = Tree(b, \varnothing) \quad \text{if } \textsc{Use}(b) = \varnothing \text{ and } b \text{ does not define a formal parameter} \\
(3) & Block(b, \mathbf{c}) & = Tree(b, \{\, Bot(\bot_k, \mathbf{c}) \,\}) \quad \text{if } \textsc{Use}(b) = \varnothing \text{ and } b \text{ defines the } k\text{-th formal parameter in } c_m \\
(4) & Call(f(\mathbf{a}), \mathbf{c}) & = Tree(f(\mathbf{a}), \{\, Paths(r, \mathbf{c}) \mid r \in \textsc{Ret}(f) \,\}) \quad \text{if } \forall c_i, c_j (i \neq j) \in \mathbf{c} \,.\, f_i \neq f_j \\
(5) & Call(f(\mathbf{a}), \mathbf{c}) & = Tree(f(\mathbf{a}), \varnothing) \quad \text{if } \exists c_i, c_j (i \neq j) \in \mathbf{c} \,.\, f_i = f_j \\
(6) & Bot(\bot_k, \mathbf{c}) & = Tree(\bot, \varnothing) \quad \text{if } \textsc{Use}(a^m_k) = \varnothing \\
(7) & Bot(\bot_k, \mathbf{c}) & = Tree(b_m, \{\, Paths(Rec(b_m, u, [c_1, \ldots, c_{m-1}])) \mid u \in \textsc{Use}(a^m_k) \,\}) \quad \text{if } \textsc{Use}(a^m_k) \neq \varnothing
\end{cases}$$

Fig. 5: Definition of the function *Paths*.

$$Rec(b, arg, \mathbf{c}) = \begin{cases} (\textsc{Rd}(b, v), \mathbf{c}) & \text{if } arg = v \in \mathcal{Vars} \\ (f(\mathbf{a}), [c_1, \ldots, c_m, (b, f, \mathbf{a})]) & \text{if } arg = f(\mathbf{a}) \in \mathcal{FunCalls} \end{cases}$$

Intuitively, if arg is a variable, *Rec* computes the RD analysis of arg from the block b looking for the variables which arg depends on, while if arg is a function call, *Rec* updates the history of function calls $\mathbf{c}$.

The tree computation algorithm starts calling $Paths(b, \varepsilon)$ for each block b containing a string-to-code statement. Let us explain the definition of $Paths(arg, \mathbf{c})$ given in Fig. 5 case by case:

$arg = b \in \mathcal{Blocks}$: $Paths(arg, \mathbf{c}) = Block(b, \mathbf{c})$

- $\textsc{Use}(b) \neq \varnothing$ (case (1)): If we reach a block b (not ground) containing one or more uses u, we create a tree with b as root, and as children all the trees resulting from calling *Paths* on all the recursive arguments (computed by $Rec(b, u, \mathbf{c})$, for each $u \in \textsc{Use}(b)$).
- $\textsc{Use}(b) = \varnothing$ (cases (2) and (3)): When we reach a ground block b we have to distinguish two cases: When the block does not define any formal parameter (of the function including it), the analysis terminates on the current block b (case (2)); Otherwise, a subtree is created with root b and children the trees resulting from the analysis of the corresponding actual parameter by calling $Bot(\bot_k, \mathbf{c})$ (case (3)).

$arg = f(\mathbf{a}) \in \mathcal{FunCalls}$: $Paths(arg, \mathbf{c}) = Call(f(\mathbf{a}), \mathbf{c})$

When *Paths* is called on a function call $f(\mathbf{a})$, and the call has been already met before in $\mathbf{c}$, then the analysis stops adding $f(\mathbf{a})$ to the tree (case (5)). Otherwise, a subtree is created with $f(\mathbf{a})$ as root, and as children all the trees resulting from calling *Paths* on all the return blocks ($\textsc{Ret}(f)$) of the function f (case (4)). The idea beyond this strategy comes from the fact that a function call can, in the worst case, only propagate, and not generate, flows.

$arg = \bot_k \in \{\bot_k\}_{k\in\mathbb{N}}$: $Paths(arg, \mathbf{c}) = Bot(\bot_k, \mathbf{c})$
We have $\bot_k$ when we reach the definition of a formal parameter (as explained in case (3)). If in the last performed call (m-th) the k-th formal parameter contains some uses ($\textsc{Use}(a_k^m) \neq \varnothing$) then we track back these uses calling *Paths* similarly to case (1) (case (7)). On the contrary, if $\textsc{Use}(a_k^m) = \varnothing$ the analysis stops adding $\bot_k$ to the tree (case (6)).

We define the set $\mathbb{T} = \{\, Paths(b_e, \varepsilon) \mid \exists C^{ssa} \in \mathbb{C}^{ssa} \,.\, b_e \in C^{ssa} \wedge b_e$ contains a string-to-code statement $\}$, i.e., the set of trees generated by the function *Paths*.

In the example in Fig. 3a, a code injection attack may be possible via the input `s`, since it interferes with the first query execution (as explained in Sect. 1), but not with second one. The tree generated by $Paths(b_1, \varepsilon)$ is depicted in Fig. 3c on the left. In this case the only leaf is the block b_0, in which an untrusted input `s` is required. This means that the corresponding path is a code injection vulnerable path. On the other hand, the resulting tree of the second query execution $Paths(b_2, \varepsilon)$ is given in Fig. 3c on the right. In this case, the path ends up in the block b_6 which is not an untrusted input, and therefore meaning that it is not a vulnerable path. Hence, in this example $\mathbb{T} = \{\langle b_1 : \langle f(s_0) : \langle b_4 : \langle b_3 : \langle b_1 : \langle b_0\rangle\rangle\rangle\rangle\rangle\rangle, \langle b_2 : \langle g(s_0) : \langle b_6\rangle\rangle\rangle\}$.

Once we have the set of trees $\mathbb{T}$ with all the paths leading from a string-to-code statement to either (i) a ground, or (ii) a bottom block or (iii) an untrusted input block, we discard all the safe paths, i.e., those not depending on untrusted inputs (cases (ii, iii)). In addition, we also need to remove non-executable blocks, i.e., those added during the algorithm computation and which do not correspond to application/code statements. These blocks are the ones related to function calls (added by *Paths* in cases (4) and (5)), which are part of the abstract syntax of others blocks and those re-added after the analysis of a function call (see case (7) in *Paths*). We define $\mathbb{T}^c$ as the set of all cleaned up trees in $\mathbb{T}$. In the running example, $\mathbb{T}^c = \{\langle b_1 : \langle b_4 : \langle b_3 : \langle b_0\rangle\rangle\rangle\rangle\}$ (Fig. 3d). Finally, it is possible to make some transformations that will make easier to dynamically associate the executed path with the right contract. In particular, we reverse the paths in order to have them in the execution direction and we define the set $\mathbb{P} = \{\, \textsc{Reverse}(p) \mid \exists T^c \in \mathbb{T}^c$ such that $p \in \textsc{Branches}(T^c) \,\}$. In the example, $\mathbb{P} = \{b_0\, b_3\, b_4\, b_1\}$.

Contracts. At this point, the programmer, for each path in $\mathbb{P}$ leading to an untrusted input x has to provide a *contract*, i.e., a decidable language $\mathcal{I}^x$ (e.g., regular, context free, etc.) of acceptable values for that path:

$$\mathcal{C}ontracts = \{\, (\mathcal{I}^x, p) \mid \mathcal{I}^x \subseteq \mathcal{V}al^x \text{ decidable input language}, p \in \mathbb{P} \,\}$$

Hence, a contract $(\mathcal{I}^x, p)$ means that the value of the untrusted input in the first block of p, i.e., p_0, has to be in the language $\mathcal{I}^x$ if all the blocks in p have

Algorithm 1 *MonInt*

```
procedure MonInt$((\mathbb{C}, \mathcal{Contracts})_P, \mu_0)$
  $V = \varnothing$
  $\mu = \mu_0$
  $b = b_0$                    ▷ initial block of the program
  while $b \neq \bot$ do             ▷ $\bot$ is the exit block
    $D_{\text{Blocks}} = D_{\text{Blocks}} \cup \{b\}$
    if $b \neq b_0$ then $D_{\text{Edge}} = D_{\text{Edge}} \cup \{(b_p, b)\}$
    if $b$ contains an untrusted input $x$ then
      $V = V \cup \{(x, \mu(x))\}$
    end if
    if $b$ executes code then
      Verify$(V, \mathcal{Contracts}, D, b)$
    end if
    $b_p = b$
    $(b, \mu) = $ INTERPRETER$(b, \mu, \mathbb{C})$
  end while
end procedure
```

Algorithm 2 *Verify*

```
procedure Verify$(V, \mathcal{Contracts}, D, b)$
  for all $c = (\mathcal{I}^x, p_0 \cdots p_n) \in \mathcal{Contracts}$ do
    if $p_n \neq b$ then continue
    for $i = 0$ to $n - 1$ do
      if not REACHABILITY$(D, p_i, p_{i+1})$ then
        continue to the next contract $c$
      end if
    end for
    $x = $ untrusted input variable in $p_0$
    if $V(x) \notin \mathcal{I}^x$ then Throw Exception
  end for
end procedure
```

Fig. 6: Monitor algorithm.

been executed. For instance, a contract $\mathcal{I}^{\mathsf{s}}$ for the reverse of path p of the tree in Fig. 3d could be expressed by the regular expression `0 | [1-9][0-9]`* to force the untrusted input `s` to be an integer.

Finally, given a program P to analyze, the static phase provides in output the pair $(\mathbb{C}, \mathcal{Contracts})_P$.

4.3 Dynamic phase

In this section, we explain how we intend to use the result of the static phase in order to provide a monitor, i.e., a dynamic checker, of potential code injection attacks. We observe that code injection is a safety property [27] since, once a string-to-code statement depends on an untrusted input at the program point of its execution, then a vulnerability definitively occurred, meaning that the only possibility for enforcing the safety property is to stop computation.

The monitor algorithm. In the following, we develop a monitor, exploiting the contracts verifier only when necessary. In particular, the idea is to design a monitor which executes directly the language interpreter on all the statements, except on string-to-code ones, for which the monitor has prior to check the satisfiability of (potentially many) contracts.

The *MonInt* procedure (Algorithm 1 in Fig. 6) takes as input the result $(\mathbb{C}, \mathcal{Contracts})_P$ of the static analysis on the program P and an initial memory μ_0. In order to determine the right contracts to check, the algorithm keeps up-to-date a dynamic structure D, in which the information of the path followed by the execution is stored: Every time a new block b is reached, the set of blocks is expanded by adding b into it (line 6), and a new edge from to previously executed block b_p to b is added to the set of edges (line 7); We will refer to

```
 1 main(s) {
 2   i := 0;
 3   copy := "";
 4   while (i <= length(s)) {
 5     c := s[i];
 6     if      (c == "a") then { copy := copy + "a" }
 7     else if (c == "b") then { copy := copy + "b" }
 8     ...
 9     else if (c == "z") then { copy := copy + "z" };
10     i := i + 1
11   };
12   eval(copy)
13 }
```

Fig. 7: Example of an implicit flow through a conditional copy.

$D = (D_{block}, D_{edge})$ as the *dynamic CFG*. Then, if the current block is the initialization of an untrusted input x, we have to store (in V) the initial value of x, that will be potentially checked in future (lines 8–10). If the current block is a string-to-code statement, the contracts verification procedure *Verify* (Algorithm 2 in Fig. 6) is called. It stops the execution if a contract is not satisfied, meaning that the program is potentially under attack (lines 11–13). Finally, the previous block b_p is updated with the current block b, and the language interpreter INTERPRETER executes the instruction associated to the block b and updates the current block and memory (b, μ) (lines $14, 15$).

The *Verify* procedure (Algorithm 2) is the contracts verifier. It is able to stop the execution by throwing an exception if it finds an input not belonging to the language specified in the contract. In particular, the verifier iterates over the set of contracts (line 2) and picks only those contracts ending in the current block b, i.e., $(\mathcal{G}^x, p_0 \dots p_m) \in \mathscr{Contracts}$ such that $p_m = b$ (line 3). Then, it checks whether the path $p_0 \cdots p_m$ is a sub-path of the dynamic CFG D: This is achieved by $m-1$ calls to REACHABILITY algorithm, returning `true` iff there is a path in D from p_i to p_{i+1} (lines 4–8) for each $0 \leq i < n$. In this case, the procedure checks whether the contract is satisfied by checking if the input value $V(x)$ of the untrusted variable in x, input in the block p_0, is in the language $\mathcal{G}^x$ (lines $9, 10$).

4.4 Handling the implicit information flows

In this section, we show how it is possible to extend the static analysis algorithm in order to track not only the explicit information flows but also the implicit ones. The *explicit information flows* are caused by a direct exchange of information via copy operations, while the *implicit information flows* arise from the control structure of the program [26]. For instance, consider the program in Fig. 7 which performs a *conditional copy* [25] of the input and then evaluates it as code. Its input/output semantic is identical to the program `main(s) { eval(s) }`, but it does not directly copy any bit of the untrusted input into the string-to-code statement (line 12). By only changing the USE function definition, we are able to detect also this kind of attacks: We define the function GUARD(b) which computes the *immediate guard* to which the block b is subjected. For instance, in Fig. 7, if we compute $\text{Guard}(\boxed{\texttt{copy := copy + "a"}})$ it returns the set $\{\boxed{\texttt{c == "a"}}\}$. Hence, we can define the extended function $\widetilde{\text{Use}}(b) = \text{Use}(b) \cup \text{Use}(\text{Guard}(b))$ which correctly detects also the implicit flow (this technique is based on the notion of *control dependences* [7]).

4.5 Soundness

In Sect. 3, we have provided a model which precisely describes safety against code injection attacks, parametrically on the expected inputs, by embedding the programmer's intention into the definition (i.e., the predicate ϕ_g). However, the semantic aspects of the model and, in particular, the abstract non-interference predicate, make it not suitable to a straightforward implementation. Nonetheless, a semantic definition is fundamental in order to provide an accurate description of the real world problem of code injection.

We now prove the soundness of the proposed approach, w.r.t. an abstract code injection policy $\text{ANI}_{\text{P}}^{x}(\mathcal{O}, n)$ we aim at enforcing in a program P. The proposed static analysis will capture any vulnerable path by over-approximating them.

Theorem 1 (Soundness of static analysis w.r.t. $\text{ANI}_{\text{P}}^{x}(\mathcal{O}, n)$**).** *Let* $P \in$ WhileFun *be a program,* x *be an untrusted input of* P*,* $n \in \mathcal{L}_P$ *such that* $\mathit{Stm}(n) \in$ *str2code and* $\text{ANI}_{P}^{x}(\mathcal{O}, n)$ *an abstract code injection policy, then*

$$\neg\, \text{ANI}_{P}^{x}(\mathcal{O}, n) \implies \exists p = b_x \cdots b_n \in \text{Reverse}(\text{Branches}(\mathit{Paths}(b_n, \varepsilon)))$$

where b_x *and* b_n *denote the blocks containing the untrusted input* x *and the string-to-code statement at program point* n *in the* `main` *procedure, respectively.*

If an untrusted input x interferes with the execution of a string-to-code statement at a program point n, the static analysis will generate a tree rooted in b_n and leading to the leaf b_x, and the programmer will have to specify a contract for the input x concerning the cleaned and reversed path $b_x \cdots b_n$. In that light, the static analysis produces the information used by the monitor to work properly.

We now prove the correctness of the dynamic phase, namely of the *MonInt* algorithm. Its semantics is straightforward: Starting from the initial memory μ_0, it executes the program until a string-to-code statement is not reached. When this happens, if all the contracts on the executed path are satisfied, the statement is executed, otherwise, the execution is stopped throwing an exception.

Theorem 2. *Let* $P \in$ WhileFun *be a program. For each initial memory* $\mu_0 \in \mathcal{Mem}$*,* $\tau = \langle\!|\, \mathit{MonInt} \,|\!\rangle_{(\mathbb{C}, \mathcal{Contracts})_P}(\langle 1^1, \mu_0 \rangle)$[5] *implies that*

- τ *is prefix of* $\tau' = \langle\!| P |\!\rangle(\langle 1^1, \mu_0 \rangle)$*;*
- $\tau = \langle\!| P |\!\rangle(\langle 1^1, \mu_0 \rangle)$ *if and only if for each string-to-code statement executed in* τ *all the contracts are satisfied.*

[5] Execution of the monitor specialized on the annotated program.

In other words, the monitor alters the semantics of the program (by blocking the execution) if and only if at least one input contract, for an untrusted input affecting an executed string-to-code statement, is not satisfied.

Finally, as a corollary to the Theorems 1 and 2, we can set out the following result that justifies our mechanism. Let us define the function associating with each trajectory the sequence of blocks executed in the CFG as $bl(\tau_0 \cdots \tau_n) = bl(\tau_0) \cdots bl(\tau_n)$, where $bl(\langle n^k, \mu \rangle) = b$ if b is the block containing $Stm(n)$.

Corollary 1. *Let $P \in$ WHILEFUN be a program, $n \in \mathcal{L}_P$ such that $Stm(n) \in$ str2code, and $(\mathbb{C}, \mathcal{C}ontracts)_P$ be the static analysis output. Let x be an input of P such that $\exists (\mathcal{I}^x, p_0 \cdots p_m) \in \mathcal{C}ontracts$ with p_0 setting the input x and p_m containing $Stm(n)$[6]. Let $v \in \mathcal{V}al^x$, we define $\rho_{\mathcal{M}}$ as*

$$\rho_{\mathcal{M}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v] \rangle)) \textit{ iff}$$
$$\forall (\mathcal{I}^x, p) \in \mathcal{C}ontracts.\, p \textit{ sub-trace of } bl(\langle\!| P |\!\rangle^{n^k}(\langle 1^1, \mu_0[x \leftarrow v] \rangle)) \textit{ we have } v \in \mathcal{I}^x$$

Then, given $\phi_{\mathcal{M}}$ is defined in terms of $\rho_{\mathcal{M}}$ as in Eq. 1, $\mathrm{MonInt}_{(\mathbb{C}, \mathcal{C}ontracts)_P}(\mu_0)$ enforces the abstract code injection policy: $\forall x$ untrusted input, $\forall v_1, v_2 \in \mathcal{V}al$

$$\phi^x_{\mathcal{M}}(v_1) = \phi^x_{\mathcal{M}}(v_2) \implies \rho_{\mathcal{M}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_1] \rangle)) = \rho_{\mathcal{M}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_2] \rangle))$$

Note that, our mechanism is not sound "as a matter of principle". It is sound w.r.t. the specification of contracts characterizing safe inputs and output behaviors.

4.6 Complexity considerations

The CFG model and the SSA form are well known code representations and can be computed in polynomial time w.r.t. the abstract syntax tree of the program. The bottleneck of the static phase is the *taint analysis*, computing *Paths* for each string-to-code statement. Unfortunately, the number of these paths could be exponentially large w.r.t. the size of the $\mathbb{C}^{ssa}$. This is due to the generality of our approach, allowing the programmer to provide a contract for each possible vulnerable path, i.e., for each triple (x, p, e) with x untrusted input, p path, and e string-to-code statement. In practice, it is highly unlikely to have an exponential number of ways in which an untrusted input can interfere with a single query execution, therefore we believe that in the average case, this approach scales well. However, the programmer can always reduce the complexity by either providing a (different) contract for each pair (x, e) (reducing complexity to $O(ne)$, n number of the untrusted inputs and e number of string-to-code blocks), or providing a different contract only for each input x (reducing complexity to $O(n)$).

Dynamic monitor worst case checking cost is divided into the cost for computing REACHABILITY between each pair of adjacent nodes (which is polynomial w.r.t. the size of $\mathbb{C}^{ssa}$) for each contract c and checking whether an untrusted input u satisfies the corresponding contract c (whose cost depends on the formalism used to model it).

[6] We consider only one variable for simplicity, but in general we may have more than one untrusted input affecting $Stm(n)$, in this case the generalization is straightforward.

5 Generality of Abstract Code Injection

In this section, we show also by means of two examples, that the definition of abstract code injection that we provide is enough general to cope with the main related works, defining specific notions of code injection [3,24] (a brief summary of these works is given in Sec. 1).

The generality of our approach allows us to detect attacks that elude the mechanisms proposed in the related works: The program given in Fig. 7 is not detected by all the works based on a copy-based taint analysis (as [24]), since that program is built on a pure semantic notion of interference. Furthermore, the flexibility of our mechanism make it more suitable to different types of code injection. For instance, consider the simple program `main(s) { eval(s) }`: all the mechanisms based on the idea of automatically detect a manipulation of the syntactic structure cannot infer nothing about the intended structure, since there are no sufficient information to "guess" the programmer's intention.

Defining code injection attacks (CIAO) [24]. From the definition of code injection attacks given in [24], we can derive the considered set of safe output behaviors $\mathcal{O}_{[24]}$, i.e., all the states reached by a code execution that does not contain any *tainted* (potentially untrusted) *code* symbol[7]. In their paper, they provide an algorithm $A(P, T, U)$[8] to precisely detect what is considered a potential attack, w.r.t. their definition of safe behaviors $\mathcal{O}_{[24]}$. Being (T, U) the tuple of all (trusted and untrusted) inputs, it corresponds to our memory μ_0. We can model their computation of safe behaviors as the characteristic function $\rho_{\mathcal{O}}^{[24]}$ of $\mathcal{O}_{[24]}$, defined as: For each execution point n^k (where a string-to-code statement is executed)

$$\rho_{\mathcal{O}}^{[24]}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \overbrace{\mu_0}^{\text{i.e., } (T,U)} \rangle)) \iff A(P, \mu_0) \text{ does not detect an attack at the point } n^k$$

Hence, the algorithm proposed in [24] enforces the abstract code injection policy: $\forall x \in U, v_1, v_2 \in \mathcal{Val}^x$

$$\phi^x_{\mathcal{O}_{[24]}}(v_1) = \phi^x_{\mathcal{O}_{[24]}}(v_2) \implies \rho_{\mathcal{O}}^{[24]}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_1] \rangle) = \rho_{\mathcal{O}}^{[24]}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_2] \rangle))$$

where $\phi^x_{\mathcal{O}_{[24]}}$ is defined in terms of $\rho_{\mathcal{O}}^{[24]}$ as in Eq. 1. Hence, from the semantic perspective of our approach, [24] only admits the (abstract) interference that does not cause the execution of tainted code symbols. This is clearly an abstract form of non-interference.

CANDID [3]. In this approach, we can still derive the implicitly used notion of safe output behaviors $\mathcal{O}_{[3]}$ as the set of all the states in which the syntactic structures (i.e., the parsing trees) of each query[9] executed by P on the inputs $i_1, \dots, i_n$, are equal to the ones produced by the execution of the program P on the *valid representation*[10] of the inputs $i_1, \dots, i_n$, i.e., $\mathrm{VR}(i_1), \dots, \mathrm{VR}(i_n)$. Let

[7] In [24], a symbol is considered *code* if it is not a final value.

[8] P is a program, and T and U are the set of trusted and untrusted inputs, respectively.

[9] [3] is focused on the SQLi problem.

[10] A *valid representation* of an input i is a value $\mathrm{VR}(i)$ which is manifestly benign and non-attacking, and it dictates the same path of i in the application.

$B(P, i_1, \ldots, i_n)$ be the function returning the syntactic structure of the query executed at the execution point n^k, we can define the characteristic function $\rho_{\mathbb{O}}^{[3]}$ of $\mathbb{O}_{[3]}$ as

$$\rho_{\mathbb{O}}^{[3]}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \overbrace{\mu_0}^{\text{i.e., } (i_1, \ldots, i_n)} \rangle)) \iff B(P, \mu_0) \overbrace{\approx}^{\text{is isomorphic to}} B(P, \mathrm{VR}(\mu_0))$$

Intuitively, in CANDID are safe all the query executions that performed on an input i or on its valid representation $VR(i)$ provide the same execution structure. As before, we can characterize the abstract code injection policy enforced by CANDID as: $\forall x \in U, v_1, v_2 \in \mathcal{V}a\ell^x$

$$\phi^x_{\mathbb{O}_{[3]}}(v_1) = \phi^x_{\mathbb{O}_{[3]}}(v_2) \implies \rho_{\mathbb{O}}^{[3]}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_1] \rangle)) = \rho_{\mathbb{O}}^{[3]}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_2] \rangle)))$$

where, again as before, $\phi^x_{\mathbb{O}_{[3]}}$ is defined in terms of $\rho_{\mathbb{O}}^{[3]}$. Here, the non-admitted interference concerns the alteration of the structure of the parsing tree.

6 Conclusion

In this paper we propose both a general model for abstract code injection policies, i.e., code injection policies parametric on what the programmer considers safe in output, and an algorithmic approach for enforcing abstract code injection policies, based on the combination of a static and a dynamic analysis phase. In particular, the static analysis aids the programmer in finding *what* should be controlled and *when*, i.e., which inputs and which execution paths, have to be checked during execution. The *contracts* that the inputs should meet are asked to the programmer and used to annotate the program. Then a monitor checking the contracts when necessary, namely when a vulnerable path is executed, is proposed. In particular, the application enforcing a given abstract code injection policy consists in the monitor specialized on the annotated program. We finally provide the intuition of the generality of abstract code injection, by showing the abstract injection policies enforced by the main related works.

We tested the feasibility of the monitoring approach by implementing it on a toy language for SQL injection[11], but surely in the future we aim at implementing this analysis approach on real languages. As far as the model is concerned, there are several aspects that deserve further study. In this paper we consider only safe output partitions in safe/unsafe behaviors, but abstract code injection policies could be defined in terms of more precise partitions, providing the possibility of modeling different safety degrees of output behaviors. Finally, in the approach we propose to enforce the policy where the output characterization of safe bahaviors is determined by the input contracts. It would be interesting to find a way for approximating input decidable contracts automatically generated by the output characterization of safe behaviors.

[11] The source code is available for the use at `https://gitlab.com/samuele/KArMA.git`.

References

1. The Open Web Application Security Project (OWASP), https://www.owasp.org/
2. Anley, C.: Advanced sql injection in sql server applications (2002)
3. Bandhakavi, S., Bisht, P., Madhusudan, P., Venkatakrishnan, V.: Candid: preventing sql injection attacks using dynamic candidate evaluations. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 12–24. ACM (2007)
4. Buehrer, G., Weide, B.W., Sivilotti, P.A.: Using parse tree validation to prevent sql injection attacks. In: Proceedings of the 5th international workshop on Software engineering and middleware. pp. 106–113. ACM (2005)
5. Cooper, K., Torczon, L.: Engineering a compiler. Elsevier (2011)
6. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theor. Comput. Sci. 277(1-2), 47–103 (2002)
7. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(4), 451–490 (1991)
8. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: Parameterizing non-interference by abstract interpretation. ACM SIGPLAN Notices 39(1), 186–197 (2004)
9. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Security and Privacy, 1982 IEEE Symposium on. pp. 11–11. IEEE (1982)
10. Halfond, W.G., Orso, A.: Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. pp. 174–183. ACM (2005)
11. Halfond, W.G., Orso, A., Manolios, P.: Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering. pp. 175–185. ACM (2006)
12. Jones, N., Sestoft, P., Søndergaard, H.: An experiment in partial evaluation: The generation of a compiler generator. In: Proc. of the 1st Internat. Conf. on Rewriting techniques and applications. vol. 202, pp. 124–140 (05 1985)
13. Maor, O., Shulman, A.: Sql injection signatures evasion. Imperva, Inc., Apr (2004)
14. Mastroeni, I.: On the rôle of abstract non-interference in language-based security. In: Programming Languages and Systems, Third Asian Symposium, APLAS. pp. 418–433 (2005)
15. Mastroeni, I.: Abstract interpretation-based approaches to security - A survey on abstract non-interference and its challenging applications. arXiv preprint arXiv:1309.5131 (2013)
16. Mastroeni, I., Banerjee, A.: Modelling declassification policies using abstract domain completeness. Mathematical Structures in Computer Science 21(6), 1253–1299 (2011)
17. Mastroeni, I., Zanardini, D.: Abstract program slicing: an abstract interpretation-based approach to program slicing. ACM Transactions on Computational Logic (TOCL) 18(1), 7 (2017)
18. McDonald, S.: Sql injection: Modes of attack, defense, and why it matters. White paper, GovernmentSecurity. org (2002)
19. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically hardening web applications using precise tainting. Security and Privacy in the Age of Ubiquitous Computing pp. 295–307 (2005)

20. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)
21. OWASP: Top 10 2010. The Ten Most Critical Web Application Security Risks (2010)
22. OWASP: Top 10 2013. The Ten Most Critical Web Application Security Risks (2013)
23. OWASP: Top 10 2017 (release candidate 1). The Ten Most Critical Web Application Security Risks (2017)
24. Ray, D., Ligatti, J.: Defining code-injection attacks. In: ACM SIGPLAN Notices. vol. 47, pp. 179–190. ACM (2012)
25. Ruse, M.E., Basu, S.: Detecting cross-site scripting vulnerability using concolic testing. In: Information Technology: New Generations (ITNG), 2013 Tenth International Conference on. pp. 633–638. IEEE (2013)
26. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on selected areas in communications 21(1), 5–19 (2003)
27. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security (TISSEC) 3(1), 30–50 (2000)
28. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: ACM SIGPLAN Notices. vol. 41, pp. 372–382. ACM (2006)
29. The Code Curmudgeon: Sql injection hall-of-shame. http://codecurmudgeon.com/wp/sql-injection-hall-of-shame/
30. Wassermann, G., Su, Z.: An analysis framework for security in web applications. In: Proceedings of the FSE Workshop on Specification and Verification of component-Based Systems (SAVCBS 2004). pp. 70–78 (2004)
31. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In: USENIX Security Symposium. pp. 121–136 (2006)