Rapporto di ricerca
Research report
April 6, 2017

102/2017

KArMA
A Knowledge-Aided Monitoring Approach for SQL Injection Attacks

Samuele Buro
Isabella Mastroeni

Abstract

In the general context of web security, there is a clear difficulty in preventing, or even modeling, SQL injection attacks due to the nature itself of this kind of attacks. They derive from the execution of an untrusted input containing data created precisely for making the program execute unintended code. It is quite hard to universally model what is *unintended*, since it may depend on the particular kind of program. In this paper, we propose a monitoring approach (KArMA - Knowledge-Aided Monitoring Approach) combining *static and dynamic analyses*. In particular, in the static phase we aid the programmer in understanding which (*untrusted*) inputs' structure need to be fixed (by *contracts*) for preventing attacks. Then, during execution, a monitor checks whether the dynamic structures respect the given contracts. In order to prove the feasibility of KArMA, we develop a prototype tool for a simplified programming language.

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. According to OWASP (Open Web Application Security Project) [1], the most critical security risks are application level injections attacks. It is worth noting that, these features, almost unavoidable, become real opportunities for the attackers which can embed malicious code which disclose and/or corrupt valuable information.

One of the main problems, due to web application injection vulnerability, is information leakage. The host used for computation, may violate security by either leaking information itself or causing other hosts to leak information [2]. Unfortunately, this is not the only possible exploitation of injection vulnerabilities, hence it becomes necessary to tackle the problem from a more general point of view, focusing on *how* a vulnerability may be exploited in an attack rather than on preventing a particular kind of payload (such as information leakage).

The main problem, that we can observe in facing code injection from a general point of view, is the difficulty of providing formal definitions of attacks. In the literature, the definitions of injection are quite specific and not really general [3, 12, 15] or, they are general enough, but less formal as, for example, the definition is given in [1] which exhaustively describe the problem of code injection:

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

The lack of formalism of this definition is mainly due to the difficulty of universally formalizing the concept of *unintended commands*. Indeed, any formalization of code injection have to fix what is *unintended* independently from the web application to analyze, potentially loosing the accuracy of the mechanism [12], when not even the completeness [3, 15] (w.r.t. the definition above).

In this paper, we embed, in the analysis process, the intention of the programmer, by generalizing the idea proposed in [3], and focusing on SQL injection [1]. What we propose is an analysis mechanism sufficiently flexible to let the programmer to tune what is intended/unintended, structured as follows:

- We first perform a *static* analysis gathering the information necessary to understand which are the program SQL injection vulnerabilities, and requiring, for each vulnerable execution path[1], *contracts* which should fix the expected untrusted input language guaranteeing prevention from SQLIA;

- We ask the programmer to characterize these contracts as a language (context free or regular language in order to guarantee decidability) of safe inputs;

- We obtain the *monitored program* which, while executing the program, performs a *dynamic* monitor, checking, at each execution point, whether the dynamic input structure respects the programmer contracts.

Moreover, if we aim at making automatic the second step, we can provide a default model of intended input, such as the ones proposed in [3, 12], and monitor the application against this model. The main advantages of the approach we propose are the following:

- This approach allows the programmers to develop web application without worrying of inserting too much checks (that could make heavier the execution). A tool based on our approach would identify the precise vulnerable points, and allow to perform checks only when necessary (during execution);

- This approach is extremely flexible from different point of views: it allows to tune security checks; the programmer may tune the complexity of the static phase by deciding whether to execute it rather than putting by hand contracts, and/or by deciding to use a fixed contract for any untrusted input rather than fixing a different contract for each input, or rather than fixing the contracts depending also on the particular path of execution; we believe it may easily applied to other forms of injection.

Why not a real implementation? We decided to implement a prototype of our approach on a toy language, rather than implementing a tool on a real language. This choice is due to our aim of providing a general approach, potentially suitable for any real context, together with the practical proof that this approach is feasible. This was not really possible if we would have provided the architecture of a tool working for a real language, since nowadays web applications are rarely developed “from scratch”. They are mainly developed by using several *frameworks* driving the implementation by abstracting several low or medium level details. There are lots of these frameworks and they are in continuous evolution, each one using different

[1]We consider vulnerable any execution path leading from an untrusted input to the execution of a query.

mechanism and providing different advantages. Hence, a real tool would have been to be developed in such a framework, and the particular choice of a framework would have forced us to analyze several details not really related to the idea we propose, making the work too specific both for understanding the main idea and for being of interest for developers using different frameworks. On the other hand, by showing the key aspects of our approach from a general point of view, we allow to implement our mechanism in any framework.

2 Background

Static Single Assignment (SSA) and reaching definition analysis. SSA [5] 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. The problem is that different definitions may reach the same use of a given identifier. In order to handle these possibilities, 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. The presence of these ϕ-functions makes the code not-executable. Consider, for instance, the following simple examples [5]:

```
V ← 4          $V_1$ ← 4          if P              if P
Z ← V + 5      $Z_1$ ← $V_1$ + 5     then V ← 4        then $V_1$ ← 4
V ← 6          $V_2$ ← 6          else V ← 6        else $V_2$ ← 6
W ← V + 7      $W_1$ ← $V_2$ + 7                      $V_3$ ← $\phi$($V_1$, $V_2$)
```

We also introduce reaching definition analysis, determining the definitions potentially reaching each use of an identifier. 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. Note that, in optimizing compiler, there exist standard techniques for reconstructing the executable program from the one in SSA form: 1) by replacing the ϕ-functions with assignment operations, and 2) by dropping subscripts [5].

The `while-fun` language and the SQL Diminished Grammar. A `while-fun` program is formed by three portions: the first and the second portion consists in all the functions and variables declarations, while the third one is the main portion of code, with all the statements. We can define untrusted inputs only in the second portion of code, by using their predefined function calls (see `E` syntactic category). The language syntax is standard, plus `execute` and function call:

$$\begin{array}{l} \mathtt{C} ::= \mathtt{skip} \mid \mathtt{end} \mid x := \mathtt{E} \mid \mathtt{C}_0;\mathtt{C}_1 \mid \mathtt{while\ B\ do}\ \{\mathtt{C}\} \mid \mathtt{if\ B\ then}\ \{\mathtt{C}_0\}\ \mathtt{else}\ \{\mathtt{C}_1\} \mid \mathtt{execute}(x) \\ \mathtt{E} ::= \mathtt{untrusted\_int()[str(),\ float(),\ bool()]} \mid v \mid x \mid op\,\mathtt{E} \mid \mathtt{E}_0\,op\,\mathtt{E}_1 \mid f(\mathtt{E}_0,\ldots,\mathtt{E}_n) \end{array}$$

The semantics is the standard one, while the statement `execute` takes a string variable and executes the SQL query it contains.

In the following, we denote by $\mathbb{V}$ the set of possible values for variables in our language (integer, string,...), and by $\sigma \in \mathbb{M} : \mathit{Var} \to \mathbb{V}$ the stores associating with each variable the actual value in the memory. We denote by $\sigma[x \leftarrow v]$ the store σ' such that $\forall y \neq x.\ \sigma'(y) = \sigma(y)$ while $\sigma'(x) = v$. Moreover, given a program P, we denote by $[\![P]\!]$ its *operational small-step (trace) semantics*, i.e., the set of all the execution traces, terminating and not terminaing.

The SQL grammar, we will refer to, is the one given in [12]. It is a simple but complete grammar inspired by the MSDN SQL Minimum Grammar.

3 Formalizing code injection as interference

In the OWASP definition cited above we read that "Injection occurs when *user-supplied data* is sent to an *interpreter* as part of a command or query". In other words, when an *untrusted input* is totally o partially *executed* for instance, as it happens in our language, as part of a query.

Let x be an untrusted input, and `Exec` be an interpretation statement (e.g., the execute of a query, or a reflection statement in a dynamic language) executing a transformation τ (potentially the identity) of the input x (τ models the transformation performed by the code between the request of the input and its execution), then by definition we have potential injection when *the execution of* `Exec`$(\tau(x))$ *depends on* x. Following the definition of dependency used in (abstract) slicing [10, 11] we formalize this situation as follows:

$$\exists v_1, v_2 \in \mathbb{V}, \sigma \in \mathbb{M}\,.\ [\![\mathtt{Exec}(\tau(x))]\!]\sigma[x \leftarrow v_1] \neq [\![\mathtt{Exec}(\tau(x))]\!]\sigma[x \leftarrow v_2]$$

We recall that this notion is the negation of a form of standard non-interference, and, exactly as it happen for non-interference, it is in general too strong, since it does not allow to really distinguish between secure and potentially insecure code. In particular, this definition says that the only secure code is the one not depending on the untrusted input, but this is in general not acceptable. For instance, also a simple password request would be insecure, since it allows or not an access depending on the inserted password, which is clearly an untrusted input. For this reason, we need to formalize code injetion, and in particular SQL injection, parametrically on what the programmer consider an *acceptable behaviour* and/or what the programmer consider an *attended input.*

Hence, let $\Pi \subseteq \mathbb{M}$ the set of all the stores that the programmer consider acceptable after the execution of the untrusted input (modeling the activity that the programmer considers attended after the execution statement), we can define the characteristic map of Π as the binary predicate

$$\rho_\Pi(\sigma) = \begin{cases} \texttt{true} & \text{if } \sigma \in \Pi \\ \texttt{false} & \text{otherwise} \end{cases}$$

Then we can weaken the definition of potential injection following the idea proposed for weakening dependencies [11], obtaining

$$\exists v_1, v_2 \in \mathbb{V}, \sigma \in \mathbb{M} \,.\, \rho_\Pi([\![\mathsf{Exec}(\tau(x))]\!]\sigma[x \leftarrow v_1]) \neq \rho_\Pi([\![\mathsf{Exec}(\tau(x))]\!]\sigma[x \leftarrow v_2])$$
$$\text{namely } \exists \sigma \in \mathbb{M} \,.\, [\![\mathsf{Exec}(\tau(x))]\!]\sigma \notin \Pi$$

At this point, given a set of attended behaviours Π, the framework of abstract non-interference [6, 9] allows us to bind the predicate, determining what is attended in output, with a predicate characterizing what is acceptable in input. In general, consider a binary predicate on the output ρ, then a binary predicate on input ϕ not leading to unattended behaviours is such that

$$\begin{array}{l} \forall v_1, v_2 \in \mathbb{V}, \sigma \in \mathbb{M} \,.\, \phi(v_1) = \phi(v_2) \Rightarrow \\ \rho([\![\mathsf{Exec}(\tau(x))]\!]\sigma[x \leftarrow v_1]) = \rho([\![\mathsf{Exec}(\tau(x))]\!]\sigma[x \leftarrow v_2]) \end{array} \quad (1)$$

Hence, ϕ models the satisfaction of contracts on inputs, specifying which inputs are acceptable in order to guarantee that only attended behaviours can be reached.

Theorem 1 *Let x be an untrusted input and $\tau(x)$ a manipulation of x such that in the program $\mathsf{Exec}(\tau(x))$ is executed, and let Π_x^τ the set of attended resulting behaviours for $\mathsf{Exec}(\tau(x))$[2]. If there exists at least one $\sigma \in \mathbb{M}$ satisfying ϕ for x (i.e., $\phi(\sigma(x)) = \texttt{true}$) such that $\rho_{\Pi_x^\tau}([\![\mathsf{Exec}(\tau(x))]\!]\sigma)$ holds (i.e., $[\![\mathsf{Exec}(\tau(x))]\!]\sigma \in \Pi_x^\tau$), then Equation 1 models safety against SQL injection attacks in $\mathsf{Exec}(\tau(x))$.*

In other words, if ϕ models an invariant property of safe untrusted input, and Eq. 1 holds, then the program is not vulnerable to SQL injection in x through the path τ, under the assumption that all the behaviour in Π_x^τ are safe.
Moreover, given an untrusted input x, a path τ and a set of attended behaviours Π_x^τ (for simplicity denoted Π), the framework of abstract non-interference suggests us how to formally characterize both the contract ϕ w.r.t. ρ_Π (denoted ϕ_Π), or the attended behaviours ρ w.r.t. the contract ϕ (denoted ρ_ϕ):

$$\phi_\Pi = \left\{ \sigma \;\middle|\; \rho_\Pi([\![\mathsf{Exec}(\tau(x))]\!]\sigma) = \texttt{true} \right\} \quad (2)$$

$$\rho_\phi = \left\{ \sigma \;\middle|\; \exists \sigma' \,.\, \phi(\sigma') = \texttt{true},\ \sigma = [\![\mathsf{Exec}(\tau(x))]\!]\sigma' \right\} \quad (3)$$

[2]Note that τ models the path leading from the input request x to the execution Exec.

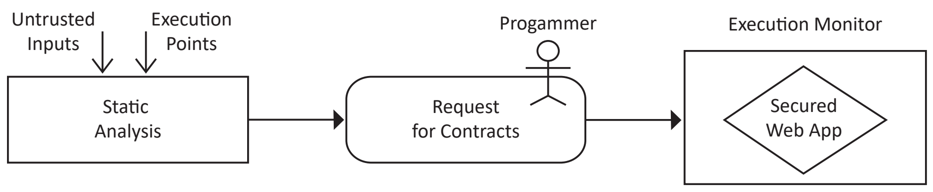


Figure 1: Defense mechanism conceptual framework.

It is clear that, the ideal situation would be to characterize ϕ_Π starting from ρ_Π, the problem is that the resulting predicate could be hard to check (it may be not decidable), hence we propose to fix a decidable ϕ (as a context free grammar or as a regular language), and to verify soundness by certifying that the corresponding attended behaviour characterization ρ_ϕ, at least, under-approximate the safe output behaviour, in order to guarantee not to miss any alarm.

Definition 1 (Soundness w.r.t. Π) *Given a contract ϕ, specifying safe untrusted input, and a set Π of safe output behaviour, characterized by ρ_Π, then ϕ is sound w.r.t. Π if $\forall\sigma.\ \rho_\phi(\llbracket \mathit{Exec}(\tau(x))\rrbracket = \mathtt{true} \Rightarrow \rho_\Pi(\llbracket \mathit{Exec}(\tau(x))\rrbracket = \mathtt{true}$.*

Hence, any contract ϕ characterizes safe behaviours w.r.t. ρ_ϕ, while it may lose precision w.r.t. more general characterization of acceptable behaviours. Contracts will model ϕ, while the monitor is the implementation of ρ_ϕ.

4 KArMA: A Knowledge-Aided Monitoring Approach

In this section, we propose a dynamic monitor able to check, and if necessary stop, the execution of an application potentially under a SQLIA. We develop two main ideas: first, we model the programmer expected structure of untrusted inputs by *contracts*; second, we combine a static analysis automatically providing *when*, during the execution, these contracts should be verified, with a monitor able to stop the computation before executing a query containing any untrusted input which do not meet the programmer intended structure. Contracts consist in a formal specification of which untrusted input (interfering with an execution) should be accepted for preventing SQLIA. In particular, this is required for each path leading from an untrusted input to the execution of a query. The approach we proposed is composed by three phases depicted in Fig. 1:

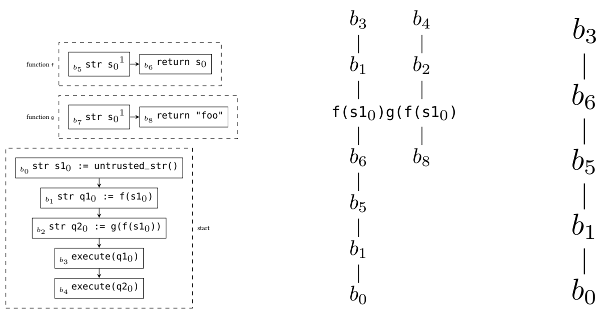
- **Static analysis:** The code is statically analyzed in order to extract the vulnerable executions, where some untrusted input interferes with an execution, and therefore where to the programmer should fix some restrictions;

```
str function f(str s) {
    return s;
};

str function g(str s) {
    return "foo";
};

str s1 := untrusted_str();
str q1 := concat("SELECT * FROM ", f(s1));
str q2 := concat("SELECT * FROM ", g(f(s1)));
execute(q1);
execute(q2);
```

(a) Program P source code.



(b) Control Flow Graph of P after the SSA form construction.

(c) Trees generated by *Paths*.

(d) Cleaned up trees.

Figure 2: Steps of static analysis algorithms.

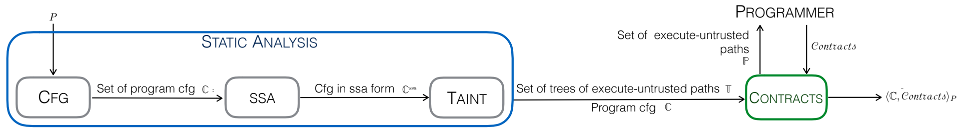


Figure 3: Static phase

- **Contracts request:** The static phase requires the programmer interaction only when potentially necessary. The contract request is an interactive phase, which could be made automatic by providing a general definition of restrictions. The result of this phase is an *annotated program* with contracts;

- **Monitor:** Finally, the monitor is able to kill the execution of an annotated program, when an untrusted input violating contracts interferes with an executed query. In other words, the result of this phase is a *monitored program* which can be seen as the monitor specialized on the annotated program.

4.1 Static phase: Analysis and contracts

In the first phase, the analysis aims at detecting any potential interference between an untrusted input and a query that can be executed. For instance, in Fig. 2(a) we should detect the potential interference between the input `s1` and the query `q1`, but not between `s1` and `q2`, since the function `g` returns always a constant value, independently from the parameter in input. The structure of the static process, including the contracts request, is depicted in Fig. 3.

Static Analysis. Let P be the program to monitor. The first two steps of the analysis consists in the construction of the program representation that will be used for performing the analysis. First, we build the control flow graph for the main program procedure ($\textsc{Cfg}_{\text{main}}$) and for each procedure declared in the program ($\textsc{Cfg}_f$) obtaining

$$\mathbb{C} = \left\{\ \textsc{Cfg}_f \ \middle|\ f \text{ procedure declared in } P\ \right\} \cup \{\textsc{Cfg}_{\text{main}}\}$$

Afterwards, in order to improve the analysis (and in particular reaching definition analysis), we consider the SSA representation of each CFG (see Sect. 2 for details), where each variable is defined only once: Let $\mathit{ssa}(\textsc{Cfg})$ be the transformer producing the $\textsc{Cfg}$ in ssa form:

$$\mathbb{C}^{\text{ssa}} = \left\{\ \mathit{ssa}(\textsc{Cfg}) \ \middle|\ \textsc{Cfg} \in \mathbb{C}\ \right\}$$

For instance, if P is the program in Fig. 2(a), $\mathit{ssa}(\textsc{Cfg}_{\text{main}})$ is given in Fig. 2(b).

Hence, we have the program representation allowing us to perform the static analysis identifying SQL injection vulnerable paths, i.e., paths where an untrusted input reaches an execution. $\textsc{Taint}$ performs, for each potentially dangerous block (in our language, any block containing the `execute` of a query) any path connecting an untrusted input with the block. In order to explain this step we have first to introduce how we represent vulnerable paths. The idea is to build, for each `execute`, a tree of all the paths leading (backwards) to an untrusted input. Hence, we define the set $\mathscr{Trees}$ of trees whose nodes are blocks, calls or the $\perp$ value, and where there is an edge from a node n_1 to a node n_2 if there is a *flow* (we will describe in details the meaning of edges when defining the semantics of $\textsc{Taint}$) from n_2 to n_1. Let n be a tree node and $\mathbf{T} \subseteq \mathscr{Trees}$, we define the tree constructor $\mathit{Tree}(n\,\mathbf{T}) = \langle n\,\mathbf{T}\rangle$ where $\langle n\rangle \equiv \langle n\,\varnothing\rangle$. Namely, starting from a set of several trees $\mathbf{T}$ we build one new tree with root n and sub-trees those in $\mathbf{T}$, i.e., there is an edge (n, m) in $\langle n\,\mathbf{T}\rangle$ for each m root of a tree in $\mathbf{T}$.
Hence, $\textsc{Taint}$, starting from the set of CFG $\mathbb{C}^{\text{ssa}}$, will return, for each `execute` statement, the tree of paths from untrusted inputs to the `execute`:

$$\mathbb{T} = \left\{\ \langle b_e\,\mathbf{T}\rangle \ \middle|\ \exists C \in \mathbb{C}^{\text{ssa}}.\ b_e \text{ is a block in } C \text{ containing } \texttt{execute}\ \right\}$$

In the example, the returned set of trees is given in Fig. 2(c).
Finally, in order to define the semantics of $\textsc{Taint}$ we need some more domains:

- $\mathscr{Fun}$ is the set of declared procedures, e.g., in the example it is $\{\mathtt{f}, \mathtt{g}\}$;
- $\mathscr{Blocks}$ is the set of all the blocks in the CFG, a block b is *ground* if it does not use any variable, i.e., $\textsc{Use}(b) = \emptyset$[3]. In the example, $\mathscr{Blocks} = \{b_0, \ldots, b_9\}$, where $\{b_0, b_5, b_7, b_8\}$ are ground;

[3]$\textsc{Use}(b)$ is the set of all the variables used in block b, where also functions calls (the function together with its list of actual parameters) are uses ($f(\mathbf{a}) \in \textsc{Use}(b)$).

- $\mathcal{FunCalls}$ is the set of functions calls $f(\mathbf{a})$, where f is the called function, while $\mathbf{a} = a_i \dots a_m$ is the set of actual parameters. In the example, it is $\{\texttt{f(sl}_0\texttt{)}, \texttt{g(f(sl}_0\texttt{))}\}$. Let us denote by $\mathcal{Args}$ the list of actual parameters of all function calls, e.g., $\mathcal{Args} = \{\texttt{sl}_0, \texttt{f(sl}_0\texttt{)}\}$.

We have now all the ingredients for describing the semantics of Taint by means of the recursive function *Paths*, which builds the potential paths of execution leading from an untrusted input to a query execution. *Paths* is a function with two parameters: The first one is a block $b \in \mathcal{Blocks}$, or a procedure call $f(\mathbf{a}) \in \mathcal{FunCalls}$, or a special value $\bot_k$ ($k \in \mathbb{N}$); The second parameter is an *history* of function calls $\mathbf{c}$. In particular, a *call* is a triple $\langle b, f, \mathbf{a}\rangle$ consisting in the calling block b, in the called function f and in its sequence of actual parameters $\mathbf{a} = a_1, \dots, a_m$. The history of calls $\mathbf{c}$ is, therefore a list of *calls* c_i, where ε denotes the empty list. Given a block $b \in \mathcal{Blocks}$ containing the `execute` of a query[4] 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 end in the executed query in b. These chains of dependencies form a tree having b as root and the blocks containing the interfering untrusted input as leaves (or bottom for the paths where surely injection attacks cannot occur).

The function *Paths*, defined in Fig. 4[5], is used to build the paths leading from an untrusted input to an execute. First, we have to define an auxiliary function used for determining the arguments of recursive calls, whose aim is that of determining the parameters of the recursive step of *Paths* and whose role is described together with *Paths*:

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

$$\mathit{Rec}(b, \mathit{arg}, \mathbf{c}) = \begin{cases} (\textsc{Rd}(b, v), \mathbf{c}) & \text{if } \mathit{arg} = v \ (1) \\ (f(\mathbf{a}), (\mathbf{c}, \langle (b, f), \mathbf{a}\rangle)) & \text{if } \mathit{arg} = f(\mathbf{a}) \ (2) \end{cases}$$

The task of *Paths* is that of identifying all the possible paths reaching `execute` from ground blocks. Unfortunately, the execution paths of a program are potentially infinite, hence the function *Paths* needs to abstract some details in order to guarantee termination. Let us describe the semantics of the call $\mathit{Paths}(b, \varepsilon)$, where b contains `execute`:

(1) $\mathit{Block}_A(b, \mathbf{c})$: If we reach a block b (not ground) containing one or more uses of variables u, then we create a tree with b as root, and as sons

[4]In our language, the only potentially dangerous statement is `execute`, it is clear that, in general, dangerous statements are language dependent.

[5]$\textsc{Ret}(f)$ is the set of all the blocks on the CFG of f containing a `return` instruction.

$$\textit{Paths}\colon (\mathscr{Blocks} \cup \mathscr{FunCalls} \cup \{\bot_n\}_{n\in\mathbb{N}}) \times ((\mathscr{Blocks} \times \mathscr{Fun}) \times \mathscr{Args})^* \to \mathscr{Tree}$$

$$\textit{Paths}(arg, \mathbf{c}) = \begin{cases} \textit{Block}_A(arg, \mathbf{c}) & \text{if } arg \in \mathscr{Blocks} \\ \textit{Call}_A(arg, \mathbf{c}) & \text{if } arg \in \mathscr{FunCalls} \\ \textit{Call}_T(arg, \mathbf{c}) & \text{if } \exists k.\ arg = \bot_k \end{cases}$$

where $\mathbf{c} = c_1 \dots c_m$, $\forall i \leq m.\ c_i = ((b_i, f_i), \mathbf{a}^i)$ and $\mathbf{a}^i = a_1^i \dots a_{n_i}^i$

$$\begin{aligned} &\textit{Block}_A \colon \mathscr{Blocks} \times ((\mathscr{Blocks} \times \mathscr{Fun}) \times \mathscr{Args})^* \to \mathscr{Tree} \\ &\textit{Call}_A \colon \mathscr{FunCalls} \times ((\mathscr{Blocks} \times \mathscr{Fun}) \times \mathscr{Args})^* \to \mathscr{Tree} \\ &\textit{Call}_T \colon \{\bot_n\}_{n\in\mathbb{N}} \times ((\mathscr{Blocks} \times \mathscr{Fun}) \times \mathscr{Args})^* \to \mathscr{Tree} \end{aligned}$$

$$\begin{cases} (1)\ \textit{Block}_A(b, \mathbf{c}) = \textit{Tree}(b, \{\textit{Paths}(\text{REC}(b, u, \mathbf{c})) \mid u \in \text{USE}(b)\}) \\ (2)\ \textit{Block}_A(b, \mathbf{c}) = \textit{Tree}(b, \varnothing) \\ \qquad \text{if USE}(b) = \varnothing \text{ and } b \text{ does not define a formal parameter} \\ (3)\ \textit{Block}_A(b, \mathbf{c}) = \textit{Tree}(b, \{\textit{Call}_T(\bot_n, \mathbf{c})\}) \\ \qquad \text{if USE}(b) = \varnothing \text{ and } b \text{ defines the } n\text{-th formal parameter of last call in } \mathbf{c} \\ (4)\ \textit{Call}_A(f(\mathbf{a}), \mathbf{c}) = \textit{Tree}(f(\mathbf{a}), \{\textit{Paths}(r, \mathbf{c}) \mid r \in \text{RET}(f)\}) \text{ if } \forall c_i, c_j.\, f_i \neq f_j \\ (5)\ \textit{Call}_A(f(\mathbf{a}), \mathbf{c}) = \textit{Tree}(f(\mathbf{a}), \varnothing) \ \text{ if } \exists c_i, c_j (i \neq j).\, f_i = f_j \\ (6)\ \textit{Call}_T(\bot_k, \mathbf{c}) = \textit{Tree}(\bot, \varnothing) \text{ if USE}(a_k^m) = \emptyset \\ (7)\ \textit{Call}_T(\bot_k, \mathbf{c}) = \\ \qquad \textit{Tree}(b_m, \{\textit{Paths}(\text{REC}(b_m, u, (c_1, \dots, c_{m-1}))) \mid u \in \text{USE}(a_k^m)\}) \\ \qquad \text{if USE}(a_k^m) \neq \emptyset \end{cases}$$

Figure 4: Definition of the function *Paths*.

all the trees resulting from calling *Paths* on all the blocks defining the variables u used in b (computed by $Rec(b, v, \mathbf{c})$);

(2)-(3) $\textit{Block}_A(b, \mathbf{c})$: When we reach a ground block b ($\text{USE}(b) = \varnothing$) we have to distinguish two cases: (2) when the block does not define any formal parameter (of the function including it), the analysis terminates on the current block b; otherwise, (3), we create a subtree with root b and sons the trees resulting by the analysis of the defined parameter, performed by $\textit{Call}_T(\bot_k, \mathbf{c})$.

(4)-(5) $\textit{Call}_A(f(\mathbf{a}), \mathbf{c})$: When *Paths* is called on a function call $f(\mathbf{a})$, and the call has been already met before in $\mathbf{c}$ (5), then the analysis stops adding $f(\mathbf{a})$ to the tree. Otherwise, (4), we create a subtree with $f(\mathbf{a})$ as root, and as sons all the trees resulting form calling *Paths* on all the return blocks ($\text{RET}(f)$) of the function f. The idea beyond this strategy comes from the fact that a function call can, in the worst case, only propagate, and not generate, flows. While, the strategy of blocking the recursive calls after the first call depends on the fact that,

$$\begin{array}{rcl}
\mathit{Paths}(b_3,\varepsilon) & = & \mathit{Tree}(\mathbf{b}_3,\{\mathit{Paths}(\textsc{Rec}(b_3,\mathtt{q1}_0,\varepsilon)\})\ (1)\\
 & & \textsc{Rec}(b_3,\mathtt{q1}_0,\varepsilon)=(\textsc{Rd}(b_3,\mathtt{q1}_0),\varepsilon)=(b_1,\varepsilon)\\
\mathit{Paths}(b_1,\varepsilon) & = & \mathit{Tree}(\mathbf{b}_1,\{\mathit{Paths}(\textsc{Rec}(b_1,\mathtt{f(s1}_0\mathtt{)},\varepsilon)\})\ (1)\\
 & & \textsc{Rec}(b_1,\mathtt{f(s1}_0\mathtt{)},\varepsilon)=(\mathtt{f(s1}_0\mathtt{)},(\langle(b_1,f),\mathtt{s1}_0\rangle))\\
 & & \text{Let } \alpha=(\langle(b_1,f),\mathtt{s1}_0\rangle)\\
\mathit{Paths}(\mathtt{f(s1}_0\mathtt{)},\alpha) & = & \mathit{Tree}(\mathtt{f(s1}_0\mathtt{)},\{\mathit{Paths}(b_6,\alpha)\})\ (4)\\
\mathit{Paths}(b_6,\alpha) & = & \mathit{Tree}(\mathbf{b}_6,\{\mathit{Paths}(\textsc{Rec}(b_6,\mathtt{s},\alpha)\})\ (1)\\
 & & \textsc{Rec}(b_6,\mathtt{s},\alpha)=(\textsc{Rd}(b_6,\mathtt{s}),\alpha)=(b_5,\alpha)\\
\mathit{Paths}(b_5,\alpha) & = & \mathit{Tree}(b_5,\{\mathit{Call}_T(\bot_1,\alpha)\})\ (3)\\
\mathit{Call}_T(\bot_1,\alpha) & = & \mathit{Tree}(\mathbf{b}_1,\{\mathit{Paths}(\textsc{Rec}(b_1,\mathtt{s1}_0,\varepsilon))\})\ (7)\\
 & & \textsc{Rec}(b_1,\mathtt{q1}_0,\varepsilon)=(\textsc{Rd}(b_1,\mathtt{s1}_0),\varepsilon)=(b_0,\varepsilon)\\
\mathit{Paths}(b_0,\varepsilon) & = & \mathit{Tree}(\mathbf{b}_0,\varnothing)=\langle b_0\rangle
\end{array}$$

Figure 5: Example of computation of *Paths*(b_3).

if a parameter interferes at a certain recursion level n, then it would interfere also at previous levels.

(6)-(7) $\mathit{Call}_T(\bot_k, \mathbf{c})$: We have $\bot_k$ when we reach the definition of a formal parameter (case (3)). If (7) in the last performed call (m-th) the k-th formal parameter contains some uses ($\textsc{Use}(a^m_k) \neq \varnothing$) then we track back these uses calling *Paths* (similar to case (1)). In this case $\mathit{Rec}(b, u, \mathbf{c})$ updates the list of function calls $\mathbf{c}$ if u is a function call, performs reaching definition otherwise. Otherwise (6) the analysis stops adding $\bot_k$ to the tree.

Let us show how *Paths* works on the code in Fig. 2(a). In the example, the execute blocks are b_3 and b_4 (in Fig. 5 we have b_3). This execution returns the tree given in Fig. 2(c), on the left. The analysis from block b_4 is similar and returns the set $\mathbb{T}$ containing only the tree given in Fig. 2(c), on the right.

Contracts. Once we have the set of trees $\mathbb{T}$ with all the paths leading from a ground or a bottom block to an execute, we need to clean up all the spurious paths, i.e., those that don't depend on untrusted inputs. In addition, we also need to remove *non-executable* blocks, i.e., the function calls blocks (because they are part of the abstract syntax of others blocks) and the blocks related to formal parameters definition (the binding between actual and formal parameters is done at the function call and not as a separated instruction block)[6]. Suppose $\mathbb{T}^c$ is the set of trees in $\mathbb{T}$ where all the trees are cleaned up, we have to provide the programmer with the set of all the

[6]These operations are strictly related on the algorithm's design choices.

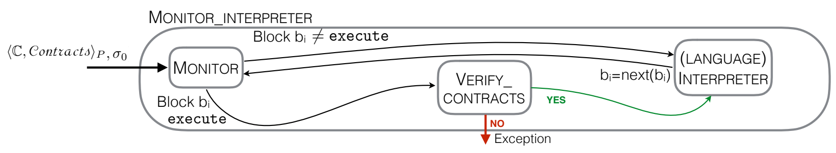


Figure 6: Monitor interpreter architecture

paths requiring a contract. For instance, in Fig. 2(d) we have the cleaned tree for the running example.

Before extracting the set of paths for the programmer, we need 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 paths in the execution direction, from inputs to `execute`.

$$\mathbb{P} = \left\{ \; reverse(p) \;\middle|\; \exists T \in \mathbb{T}^c .\, p \text{ path in } T \;\right\}$$

Proposition 1 *Let P be a program to analyze. If an untrusted input x required in block p_0 interferes with an* `execute` *statement in block p_n, for some execution of P, then there exists a path $\mathrm{p}_0 \ldots \mathrm{p}_n$ in $\mathbb{P}$ leading from p_0 to p_n.*

At this point, the programmer, for each path (identifying the pair untrusted input/execution) has to provide the contract, i.e., the grammar ϕ defining the sub-language of inputs admitted for that path:

$$\mathcal{C}ontracts = \left\{ \; \langle \phi, p \rangle \;\middle|\; \phi \subseteq \mathcal{L}(G), p \in \mathbb{P} \;\right\}$$

Hence a contract $\langle \phi, p \rangle$ means that the untrusted input required in the first block of p, i.e., p_0, has to be in the language ϕ if the last block of p is executed. For instance, a contract ϕ for the path p in Fig. 2(d) could be `INT_LITERAL` to force the untrusted input s_0 to be an integer. Finally, given a program P to analyze, the static analysis module provide in output the pair $\langle \mathbb{C}, \mathcal{C}ontracts \rangle_P$ formed by the set of CFG $\mathbb{C}$ of P and the set of contracts $\mathcal{C}ontracts$.

4.2 Dynamic phase: Monitor

In this section, we explain how we mean to use the result of the previous static analysis in order to provide a monitor, i.e., a dynamic checker, of injection vulnerabilities. It is well known that there exists a monitor only for safety properties [13]. Then we observe that SQLIA is a safety property since, once a SQL query interfers with an untrusted variable, whose structure differs from the expected one, then for the given notion of SQLIA, a vulnerability definitively occurred, meaning that allowing the computation to continue cannot lead back the system to a secure state, where the variable has the intended structure.

Algorithm 1 *Monitor_int*

```
procedure $Monitor\_int(\langle \mathbb{C}, \mathcal{Contracts} \rangle, \sigma_0)$
  $V = \varnothing$
  $\sigma = \sigma_0$
  $b = b_0$            ▷ initial block of the program
  while $b \neq \bot$ do
    $D_{\text{Blocks}} = D_{\text{Blocks}} \cup \{b\}$
    if $b \neq b_0$ then $D_{\text{Edge}} = D_{\text{Edge}} \cup \{(b_p, b)\}$
    if $b$ is an execute then
      $Verify\_con(V, \mathcal{Contracts}, D, b)$
    end if
    $b_p = b$
    $(b, \sigma) = Interpreter(b, \sigma, \mathbb{C})$
    if $b$ is an untrusted input request $x$ then
      $V = V \cup \{(x, \sigma(x))\}$
    end if
  end while
end procedure
```

Algorithm 2 *Verify_con*

```
procedure $Verify\_con(V, \mathcal{Contracts}, D, b)$
  for all $c = (\phi, p_0 \dots p_n) \in \mathcal{Contracts}$ 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
    $u =$ untrusted input variable in $p_0$
    if $V(u) \notin \phi$ then Throw Exception
  end for
end procedure
```

Figure 7: Knowledge-Aided Monitor algorithm

Algorithm for ***Monitor_int*****.** In the following, we develop a monitor, exploiting the contracts verifier only when necessary. In particular the idea is to design a monitor which, as shown in Fig. 6 executes directly the language interpreter on all the statements, except those executing a query, for which the monitor has prior to check the satisfiability of one or more contracts, i.e., of the contracts on untrusted inputs involved in the executed query. The monitor algorithm is given in Fig. 7. In Algorithm 1, the monitor takes in input the set $\mathbb{C}$ of CFG of a program P, the set of contracts $\mathcal{Contracts}$ and an initial memory σ_0 for the execution of P. First of all (lines 6-7) the process has to build the dynamic CFG D, namely it has to follow on the CFGs the paths executed considering also call and return edges between CFG of different procedure. This is used for keeping trace of the executed path, and will be used for determining the contracts to check. Then, if the current block is an untrusted input x (lines 13-14), we have to store (in V) the initial value, that will be potentially checked, of x. If the current block is the execute of a query (lines 8-9) then we launch the contract verification (Algorithm 2). Finally, if the verification procedure does not stop the execution, the current block is executed (line 12) by the language interpreter $Interpreter : \mathcal{Blocks} \times \mathbb{M} \times \wp(CFG) \to \mathcal{Blocks} \times \mathbb{M}$, and the blocks variables, the current block b and the previous executed block b_p are updated (lines 11-12), in particular b is updated with the next block to execute in $\mathbb{C}$. Algorithm 2, verifies contracts and if it finds an input not satisfying its contract it stops execution throwing an exception. In particular, it takes the executed (dynamic) CFG D and checks, for each contract $c = \langle \phi, p_0 \dots p_n \rangle$, whether the path $p_0 \dots p_n$ is a sub-path in D^7, in this case the procedure checks whether the input value $V(u)$ of the untrusted variable u required in input in the block p_0 satisfies the contract ϕ. If the contract is satisfied the execution continues, otherwise it is stopped.

The monitor semantics. Let P be the program to monitor. Let $\langle \mathbb{C}, \mathcal{Contracts}\rangle_P$ be the output of the static analysis applied to P, let $\mathcal{M} : \mathcal{Blocks} \times \mathbb{M} \times \wp(CFG) \times \mathcal{Contracts} \to (\mathcal{Blocks} \times \mathbb{M}) \cup \{\mathit{Exception}\}$ the function executing each step (block b_i) as it was the language interpreter *Interpreter*, the only difference is in the interpretation of `execute` [4, 7] which can stop the execution if a contract is not satisfied, i.e., if $\mathit{Verify}(b_i)$ returns `false`.

$$\llbracket \mathit{Monitor\_int} \rrbracket(\langle \mathbb{C}, \mathcal{Contracts}\rangle_P, \sigma_0) = \mathcal{M}(b_0, \sigma_0, \mathbb{C}, \mathcal{Contracts})$$
$$\mathcal{M}(b_i, \sigma_i, \mathbb{C}, \mathcal{Contracts}) = \begin{cases} \mathcal{M}(\mathit{Interpreter}(b_i, \sigma_i), \mathbb{C}, \mathcal{Contracts}) \\ \qquad \text{if } b_i \neq \texttt{execute} \ \vee \ \mathit{Verify}(b_i, \mathcal{Contracts}) \\ \mathit{Exception} \qquad \text{otherwise} \end{cases}$$

Theorem 2 *Let P be a program to monitor. For each initial memory $\sigma \in \mathbb{M}$ we have that $\forall t \in \llbracket \mathit{Monitor\_int} \rrbracket(\langle \mathbb{C}, \mathcal{Contracts}\rangle_P, \sigma) \exists t' \in \llbracket P \rrbracket(\sigma). t$ is prefix of t'.*

PROOF. [Scketch] $\llbracket \mathit{Monitor\_int} \rrbracket(\langle \mathbb{C}, \mathcal{Contracts}\rangle_P, \sigma) = \mathcal{M}(b_0, \sigma, \mathbb{C}, \mathcal{Contracts})$ by definition. Hence, by definition, for each step of computation it is executed $\mathit{Interpreter}(b_i, \sigma_i)$ if b_i is not an `execute`, or the corresponding contract is satisfied, otherwise the computation is killed. On the other hand each step of computation of P performs always $\mathit{Interpreter}(b_i, \sigma_i)$. Hence, the two computations are the same or the monitor has finished. □

Corollary 1 *Let P be the program to monitor, $\sigma \in \mathbb{M}$. For each `execute` in t, if it depends on some untrusted input then the input satisfies the corresponding contract iff $(t \in \llbracket \mathit{Monitor\_int} \rrbracket(\langle \mathbb{C}, \mathcal{Contracts}\rangle_P, \sigma) \Leftrightarrow t \in \llbracket P \rrbracket)$.*

Our idea is to release the monitor interpreter *Monitor_int* specialized on the program P, namely specialized on its static analysis output $\langle \mathbb{C}, \mathcal{Contracts}\rangle_P$: $\llbracket \mathit{Monitor\_int}_{\langle \mathbb{C}, \mathcal{Contracts}\rangle_P} \rrbracket(\sigma) = \llbracket \mathit{Monitor\_int} \rrbracket(\langle \mathbb{C}, \mathcal{Contracts}\rangle_P, \sigma)$

5 Analyzing and discussing feasibility of KArMA

KArMA is a *proof of concept* implementing all the proposed algorithms. Its purpose is to test our approach against `while-fun` web applications to prove the feasibility and the scalability (in terms of complexity) of the mechanism.

[7] This is achieved by $n-1$ calls to REACHABILITY algorithm, which returns `true` iff there is a path in the graph D from p_i to p_{i+1}.

Approach complexity. 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, for each `execute`, the function *Paths* (fig. 4). Unfortunately, the number of these paths (and therefore the number of branches) could be exponentially large in the size of the $\mathbb{C}^{ssa}$. This potential explosion is due to the generality of our approach, allowing the programmer to provide a, potentially different, contract for each possible vulnerable path, i.e., for each triple $\langle$Untrusted input, Path, `execute`$\rangle$. In practice, it is high unlikely to have an exponential number of ways in which an untrusted input can interfere with a single query execution, and therefore we believe that in the average case, this approach scales well on the $\mathbb{C}^{ssa}$ size. However, the programmer can always decide to reduce the complexity by (1) providing a (different) contract for each pair $\langle$Untrusted input, `execute`$\rangle$, or (2) providing a different contract depending only on the input.In the first case, the complexity of the approach is reduced to $O(ne)$ where n is the number of the untrusted inputs and e is the number of `execute` blocks while, in the second one, the complexity is further reduced to $O(n)$ (see table 1).

On the other hand, dynamic monitor checking costs $O(1)$ when a non-`execute` instruction is executed. Otherwise, the cost is divided in (1) computing Reachability between each pair of adjacent nodes (which is polynomial in the size of $\mathbb{C}^{ssa}$) for each contract c and (2) checking whether an untrusted variable initial input u satisfies the corresponding contract ϕ (which costs $O(|u|^3|G^{\phi}|)$ where $|G^{\phi}|$ is the size of the CNF grammar that generates the contract language ϕ).

The prototype. KArMA is entirely written in Java (compiled with `openjdk version "1.8.0_111"`) and implements the `while-fun` language, the static analysis and the monitor-interpreter. It receives, in input, the program source code, and it performs all the analyses ensuring that the program is well-typed. To carry out the lexical and syntactical analyses, the tool benefits from two Java libraries: the lexical analyzer JFlex [8] and the CUP [14] parser generator. KArMA implementation builds single-statement blocks: this require more time and space during the process but simplifies subsequent analysis.While computing the control flow graph, the tool derives the structures used to produce the semipruned SSA form (such as the dominator trees and the dominance frontiers) which optimizes the number of ϕ-functions placed by the algorithm. The source code of KArMA is available at `https://gitlab.com/samuele/KArMA.git`. Here it is possible to try the tool also on some examples.

Related works. In the last years, many researchers have studied mechanisms for mitigating injection attacks, while only few works focused on a

What to fix		Complexity
$\forall\ p, e, x$ .fix (p, e, x)	if x interfere with e on path p	$O(\|V\|!)$
$\forall\ e, x$. fix (e, x)	if x interfere with e	$O(ne)$
$\forall\ x$. fix x	if $\exists e$. x interfere with e	$O(n)$

Table 1: Scaling the complexity (p: path, e: `execute`, x: untrusted input).

formal definition of the problem [3, 12]:

1. [3], (`CANDID`), uses *parsing trees* for detecting syntactic differences between queries. The idea is that of dynamically characterizing the intended structure of a query obtained by running an application A on a *valid representation (VR)* of a *candidate input*, and then comparing the syntactic structure of the output program, so far obtained, with the one derived at run-time;

2. In [12], the authors split symbols in *code* and *non code*, and then they apply a taint analysis for detecting the query tainted by untrusted data. If at least one of the injected symbols in a query is code, then an SQLIA is detected, otherwise the execution proceeds. Intuitively, the non code symbols are the closed symbols of the language, since they are the irreducible elements. In general, a symbol is closed if it is a final value (such as an integer or a string).

It is worth noting that these approaches are particular instantiations of the general OWASP definition: they fix *a priori* what is *unintended* and what is not, regardless of programmer intention. In this sense, our approach is more general.

References

[1] Open Web Application Security Project (OWASP). https://www.owasp.org, 2016. [Online; accessed 13-November-2016].

[2] M. Balliu and I. Mastroeni. A weakest precondition approach to robustness. *LNCS Transactions on Computational Science*, 10:261–297, 2010.

[3] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 12–24, 2007.

[4] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 178–190, 2002.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[6] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '04)*, pages 186–197. ACM-Press, 2004.

[7] R. Giacobazzi and I. Mastroeni. Making abstract models complete. *Mathematical Structures in Computer Science*, 26(4):658–701, 2016.

[8] JFlex Team. JFlex The Fast Scanner Generator for Java — JFlex, JFlex 1.6.1.

[9] I. Mastroeni. Abstract interpretation-based approaches to security - A survey on abstract non-interference and its challenging applications. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013.*, pages 41–65, 2013.

[10] I. Mastroeni and D. Zanardini. Data dependencies and program slicing: From syntax to abstract semantics. In *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'08)*, pages 125 – 134. ACM Press, 2008.

[11] I. Mastroeni and D. Zanardini. Abstract program slicing: an abstract interpretation-based approach to program slicing. *ACM Transactions on Computational Logic*, 18(1), 2017.

[12] D. Ray and J. Ligatti. Defining code-injection attacks. In J. Field and M. Hicks, editors, *POPL*, pages 179–190. ACM, 2012.

[13] F. B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

[14] G. Scott E. Hudson, GVU Center. LALR Parser Generator for Java — CUP, CUP 0.11b.

[15] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 372–382, 2006.