A theory of injection-based vulnerabilities in formal grammars

Eric Alata[a], Pierre-François Gimenez[b]

[a] *Univ de Toulouse, INSA, LAAS, 7 avenue du colonel Roche, Toulouse, France*

[b] *CIDRE, CentraleSupélec, Inria, IRISA, Avenue de la Boulaie, Cesson-Sévigné*

Abstract

Many systems are controlled via commands built upon user inputs. For systems that deal with structured commands, such as SQL queries, XML documents, or network messages, such commands are generally constructed in a "fill-in-the-blank" fashion: the user input is concatenated with a fixed part written by the developer (the template). However, the user input can be crafted to modify the command's semantics intended by the developer and lead to the system's malicious usages. Such an attack, called an injection-based attack, is considered one of the most severe threat to web applications. Solutions to prevent such vulnerabilities exist but are generally ad hoc and rely on the developer's expertise and diligence. Our approach addresses these vulnerabilities from the formal language theory's point of view. We formally define two new security properties. The first one, "intent-equivalence", guarantees that a developer's template cannot lead to malicious injections. The second one, "intent-security", guarantees that every possible template is intent-equivalent, and therefore that the programming language itself is secure. We thoroughly analyze the decidability of these properties for the most common grammar classes. We conclude by highlighting the technical implications of these results for various settings and tools.

Keywords: formal language, injection attack, web application, security, vulnerability

1. Introduction

The multiplication of new programming languages, software engineering frameworks, and tools has significantly shortened the Internet and web-based applications' development cycles. The architectures of the applications generally share the same design pattern for accessing external resources: they rely on building one or several statements (generally called queries) for interpreters that perform the required actions on application resources. Such queries can be SQL queries for databases, shell system commands, XPath queries on XML documents, or a specific message format for a remote service on the Internet. Due to the low-level interaction between the application and the interpreter, such queries generally consist of a concatenated sequence of characters that may involve user-supplied data. More precisely, the developer generally writes a fill-in-the-blanks-type query, and user-supplied data is placed inside each "blank".

However, user input is not always legitimate. There is a whole class of attacks that aim specifically at this widespread architecture: injection-based attacks. These attacks consist of submitting malicious inputs and building a malicious query whose semantics is different from what the developer had in mind.

Email addresses: `eric.alata@laas.fr` (Eric Alata), `pierre-francois.gimenez@centralesupelec.fr` (Pierre-François Gimenez)

This vulnerability is not new: they have been known since 1998 when Phrack magazine described the first occurrence [1]. We can even retrospectively find it in phone phreaking during the 1950s. They can produce dramatic effects depending on the type of service, and their persistence vulnerability is generally attributed to a lack of developer awareness or bad practices [2]. In 2020, CWE (Common Weakness Enumeration) [3] listed SQL injection, OS-command injection, and code injection among their 25 most dangerous software vulnerabilities. OWASP (Open Web Application Security Project) ranked injection as the most dangerous web application threat in 2017 [4] and as the third most dangerous one in 2021 [5]. Indeed, injections are ubiquitous:they affect nearly all programming languages families: query languages and network protocols like SQL, even with object-relational mapping (ORM), NoSQL, HTTP, SMTP and LDAP[1], interpreted languages like JavaScript, HTML, CSS, python, Windows command line and Bash[2], compiled languages such as Java and Go[3], structured format as URL, JSON and XML[4]. This list is not exhaustive and gets longer and longer as new languages, protocols, and formats are developed and used.

Although many tools exist to identify and prevent injection-based attacks, they generally only treat the symptoms of these attacks, with ad hoc methods, and not the root cause of the vulnerability. Only a few articles were interested in the vulnerabilities embedded in the programming languages design itself, and this is the line of work we want to expand. By answering why all programming languages seem vulnerable to injection-based attacks, we seek to answer how such vulnerabilities could be detected and prevented at two steps: the design of a new language and its use by a developer. To this extent, this work relies on a new formalization of injection vulnerabilities. As suggested by [6], we consider that the developer had an intent in mind while writing the query. Any user input that this intent cannot explain is considered malicious. This new definition, particularly adapted to a theoretical analysis, led to several results on malicious injections depending on the query language grammar class. More precisely, we introduce two security properties: intent-equivalence and intent-security. A template is intent-equivalent to some intent if that intent can explain all injections, i.e., there are no malicious injections. A grammar is intent-secure if all its possible templates are intent-equivalent to some intent. In other words, an intent-secure grammar is not vulnerable to injection-based attacks. We analyze these properties for one or more blanks in the query and various grammar classes. We notably study regular, deterministic, and context-free grammars, along with some results concerning LR(k) grammars and visibly pushdown grammars. Furthermore, we characterize the set of malicious injections for various grammar classes.

Besides, our analysis also sheds new light with nuanced positions on several research questions on languages security:

- **Q1:** Does string concatenation necessarily lead to injection vulnerabilities?
- **Q2:** Could a skilled developer write only secure queries?
- **Q3:** Is a simpler language always more secure than a more complex one?
- **Q4:** Why do (nearly) all computer languages contain injection vulnerabilities?
- **Q5:** Could existing programming languages be fixed or should new programming languages be developed?

The paper is structured as follows. Section 2 discusses related work and positions our main contributions. Then, Section 3 presents some fundamental background concepts of language theory. Our fundamental

[1]HTTP: CVE-2019-13143, SQL: CVE-2019-1010259 and CVE-2019-14900; NoSQL: CVE-2021-22911; SMTP: CVE-2017-9801; LDAP: CVE-2019-4297

[2]JavaScript: CVE-2019-1020008; HTML: CVE-2019-1010113; CSS: CVE-2020-16254; python: CVE-2019-10633; Windows command line: CVE-2019-15599; Bash: CVE-2014-6271

[3]Java: CVE-2018-16621; Go: CVE-2020-28366

[4]URL: CVE-2022-0391, JSON: CVE-2018-14010; XML: CVE-2021-21025

assumptions and their technical implications are discussed in Section 4. The intent-equivalence property is defined and studied in Section 5. The intent-security property is defined and analyzed in Section 6. Then, the characterization of the unintended injections is explored in Section 7. Finally, Section 8 summarizes our major results, presents their technical implications by providing part of the answers to the above questions, and discusses future directions.

2. Related work

2.1. Protection approaches

There are four main approaches to mitigate injection-based attacks: user input processing, intrusion detection, fuzzing, and static analysis.

User input processing is a broad category of run-time protection between the reception of the user input and the sending of the query to the interpreter. It includes filtering protection that modifies the user-provided data to avoid concatenation with malicious inputs. Typical filters are character escape (for example, `&` is often escaped into `&amp;` in HTML) and keyword filtering (for example, `<script>` tags are often removed entirely in HTML). Yet, the filtering effectiveness depends on the ability to identify malicious inputs while still allowing legitimate inputs. Another example is prepared statements, well-known for SQL queries, where the developer specifies what type of token the user input should be (an integer, a string, etc.). The prepared statement library checks the user input against the expected token type and handles the concatenation if possible. However, user input processing generally needs to be applied for each template. Attacks may still occur if the developer is unaware of injection-based attacks, does not use the user input processing properly due to a lack of skills or guidance, or simply forgets to protect one template.

Intrusion detection systems (IDS) [7] are generally deployed into the communication to analyze the network traffic and flag any suspicious activity. Such systems can either be signature-based or behavior-based. Signature-based IDS includes a list of signatures (written by experts or learned by machine learning) to recognize attacks. Behavior-based IDS know the system's normal behavior (either through specification or machine learning) and raise an alert whenever there is a significant deviation. However, signature-based IDS cannot detect new attacks, and behavior-based IDS generally produce too many false positives. They are usually commercial off-the-shelf software, i.e., ready-to-use and generic components. Therefore, they are effective in detecting common malicious injections (typically with a regular expression describing these attacks), but they may miss more exotic malicious injections that subtly change the semantics of queries.

Application testing and verification using fuzzing approaches is a simple way to identify vulnerabilities for applications with no or weak filtering [8]. These tools rely on sending inputs to the target architecture and detecting abnormal behaviors. In its most straightforward flavor, it consists of sending randomly generated inputs. This basic setting has been refined considering vulnerability scanners: they use smart algorithms to select candidate inputs most likely to detect vulnerabilities. Most of the time, they are based on a knowledge of how developers use interpreters. However, as illustrated in several experimental studies, these tools' detection coverage and efficiency still need to be significantly improved [9, 10, 11, 12]. In addition, these approaches are very dependent on the level of expertise of their user. Thus, they have the same weakness as the IDS: they may miss malicious injections that subtly change the semantics of queries and require a good knowledge of the underlying language.

Static analysis methods are used to identify specific patterns corresponding to injection vulnerabilities [13, 14]. These techniques are sometimes supported by formal methods that guarantee that only malicious injections are produced or, on the other hand, that no malicious injections are ignored during the analysis. Both cases require complete knowledge of the architecture, including the interpreter. These techniques are difficult to scale by considering all possible inputs: either they require a prohibitive execution time or generate an unrealistically large set of candidate inputs.

2.2. Tools

Recent surveys [15, 16, 17, 18] related to injections confirm that most of the research in this area focuses on designing automated tools to address these vulnerabilities. The proposed tools are mainly based on run-time analysis [14, 19] or propose technical countermeasures. It is noteworthy that formal methods-based analyses or research that studies injections from a purely theoretical perspective are not referenced in these surveys.

Research strongly focuses on the SQL language, limiting the scope of the proposed methods. Some works tackle injection attacks for other languages: mainly NoSQL [20, 21] and HTML [22, 23]. It is not easy to know to what extent these methods apply to other languages. Nevertheless, it is interesting to note that many approaches have taken advantage of machine learning to detect SQL injections in the last few years ([24] and [25]). For instance, SQLBlock [25] performs a learning process on the legitimate requests observed during the controlled execution of a PHP server. Then, it instruments the interface between PHP and SQL by inserting a plugin whose role is to make sure that the queries then generated comply with the form of the legitimate queries.

Besides, tainting and instrumentation of the code are still widely used [26, 27, 28], as well as comparison with an input signature base corresponding to injections. For example, the tool of [29] is inserted between the client and the server to compare user input with an SQL injection attack database using the Baeza-Yates–Gonnet algorithm. Likewise, the solution of [30] lies between the client and the server to analyze the packets exchanged and compare their content with a database of injection attack signatures. An alternative is AUTOGRAM [31], a fuzzer that relies on dynamic tainting to infer a context-free grammar of the input of a program. However, as shown later, such an input grammar is generally not context-free in the context of injections.

In [32], a framework is deployed in the cloud to analyze network traffic and detect SQL injection attacks using regular expressions. The choice of regular expressions is relevant to optimize the processing speed but at the expense of precision. A similar choice was made by [33], [34], [35] and [36].

In [37], authors observe that no solution can be easily transferred into the industry. They reviewed different tools to deal with SQL injection attacks (SQLProb [38], SQLrand [39], CANDID [6], WASP [28], etc.) to identify the ones that have the most potential for this transfer and concluded on WASP after an empirical study.

2.3. Language-theoretic approaches

Since injection-based attacks are based on textual data, many researchers applied results from the formal language theory. This theory studies the syntax of languages and their relations with automata (theoretical computing devices) and formal grammars (succinct descriptions of a language). This approach is notably pursued by the LangSec community that uses formal language theory to propose new tools and identify anti-patterns developers should avoid. However, even if several formal definitions of injection-based vulnerabilities have been proposed [19, 6, 40], no in-depth theoretical study has been carried out yet. Such a study could lead to new tools, e.g., to support query certification (guarantee that no malicious injection is possible), create new languages without these vulnerabilities, and automatically infer the language of malicious injections associated with a query. More generally, this theoretical analysis could lead to solutions that do not require (or require much less) expertise and would not be tied to a particular language or technology. This in-depth analysis based on formal language theory is the contribution of this paper.

We mentioned in the previous subsection that regular expressions had been used to detect SQL injection attacks. However, 15 years ago, the authors of [41] have shown that detection can require a more complex model than a regular expression. This last article comes from the language-theoretic security (LangSec) community that studies input attacks from a formal point of view. These attacks notably include buffer overflow and injection attacks. In 2005, [42] used formal language results, and notably undecidability theorems, to analyze input validation and propose multiple observations and advice to developers. In [43], the authors propose to modelize input attack as a difference between the message a source wants to transmit and the message the destination receives. More precisely, they focus on an encoding function (used by the source to encode the message) and a decoding function (used by the destination to decode the encoded message). In this framework, an injection is defined as a message whose semantics is modified by the successive application of the encoder and the decoder. This framework led to several software, like McHammerCoder [44] or Nail [45].

The contributions [46, 2] of the LangSec community are complementary to our work. More precisely, this community tackles injection-based vulnerabilities from an end-to-end perspective, including the implementation issues, i.e., the difference between the target formal grammar and the parsing and unparsing software, with a particular focus on parsing flaws. For this reason, the contributions of the LangSec community generally don't rely solely on formal language theory and don't propose a formal definition nor study their theoretical implications. We consider that injection-based vulnerabilities ultimately stem from the formal grammar itself, and this is why we restrict our scope to its in-depth study.

2.4. Injection-based vulnerability formalization proposals

To the best of our knowledge, formal studies of injections have seldom been explored. The existing formal definitions are not used to characterize injections or answer the research questions raised in the introduction. Nevertheless, it is interesting to examine the scope and limitations of the proposed definitions.

The authors of [19] consider that a valid input from the user must correspond to structurally complete data (for example, a table name or a Boolean expression). They use this definition to add markers in SQL queries to delimit user inputs and test the validity of their structure. Although natural, this definition is too limited from our point of view. Indeed, for some applications, the user may be asked to provide data that is not structurally complete (for example, part of the name of a table instead of the full name or simply a Boolean operator instead of a Boolean expression).

In [6], authors focus on SQL and consider an injection to be a user input that does not belong to a finite list of symbols. Knowing the SQL query to protect and the list of legitimate symbols, they can build a list of legitimate abstract syntax trees. During the execution, the concatenation with the user data must lead to one of these abstract syntax trees considered legitimate. This definition is also debatable because the legitimate user inputs can be infinite (for example, if they correspond to a Boolean expression, given that the set of Boolean expressions is infinite). Besides, in some cases, an injection may change how the query is parsed while preserving the syntax tree's overall structure.

The last formalization we mention is not dedicated to injection-based attacks but should encompass any computer attacks. This formal framework for security uses "weird machines", a concept used by exploit practitioners and described in [47]. In this framework, what the developer intends to implement is modelized as an "intended finite state machine" (IFSM). It may differ from the actual implementation, modelized by another finite state machine that may have additional states (called "weird states") with no equivalent states in the IFSM. This framework encompasses a large number of vulnerabilities, both hardware and software. However, we argue that this framework is not adapted to injection vulnerabilities. Injection attacks typically happen when two systems interact: an interpreter that processes queries and a client that sends queries. Two disjoint teams generally develop these systems. However, in injection attacks, both systems behave accordingly to the intent of their developers. In the case of an SQL injection, for example, the interpreter receives a malicious query. Since an interpreter's purpose is to execute queries, an SQL attack can happen even if the interpreter has no weird state (i.e., is safe in the weird machine framework). It is also the case for the client. In most cases, the only difference between a legitimate and a malicious user input is in the data manipulated (generally as string). A malicious user input doesn't need to change the flow of the program (i.e., the path in the intended finite state machine). For example, an attack could affect only the interpreter system with an injected "DROP table" instruction that deletes part of a database with no (immediate) effect on the client. Injection attacks are challenging to study because they may happen at the interface of two systems with no weird states.

3. Background

This section aims at providing a brief presentation of some notions of formal language and grammar used in the next sections.

3.1. Formal grammars

A formal grammar $G = (N, T, R, S)$ consists of four elements. T is a finite set of terminal symbols, called the alphabet. N is a finite set of nonterminal symbols, disjoint from T. We will denote $\Delta = N \cup T$ the set of all symbols. The elements of Δ^* are sentential forms, and the elements of T^* are words, where * represents the Kleene star operation. The empty sentential form is denoted ϵ. The length of a sentential form α, denoted $|\alpha|$, is the number of symbols it is composed of (so $\alpha \in \Delta^{|\alpha|}$).

R is a finite set of rules (also called productions). A production is an expression of the form $\alpha \to \beta$, where $\alpha \in \Delta^+ - N^+$ (i.e., α contains at least one terminal), $\beta \in \Delta^*$. Finally, S is a symbol of N called the axiom.

In the following, we generally use uppercase Latin letters at the start of the alphabet (A, B, …) for nonterminals, lowercase Latin letters at the start of the alphabet (a, b, …) for terminals, lowercase Latin letters at the end of the alphabet (u, v, …) for words and Greek letters (α, β, …) for sentential forms.

Let $r = \alpha \to \beta$ be a rule of G. A sentential form $\omega_1\alpha\omega_2$ directly derives $\omega_1\beta\omega_2$ by applying r, written $\omega_1\alpha\omega_2 \Rightarrow_G \omega_1\beta\omega_2$, for any $\omega_1, \omega_2 \in \Delta^*$. If ω_1, ω_2, …, ω_n are sentential forms such that $\omega_1 \Rightarrow_G \omega_2 \Rightarrow_G \ldots \Rightarrow_G \omega_n$, then ω_1 derives ω_n, written $\omega_1 \Rightarrow_G^* \omega_n$ ($\Rightarrow_G^*$ is the reflexive transitive closure of $\Rightarrow_G$). Since $\Rightarrow_G^*$ is reflexive, every sentential form $\alpha \in \Delta^*$ derives itself: $\alpha \Rightarrow_G^* \alpha$. For simplicity, we will omit the subscript G when it is understood. A word w is derivable from the grammar if there exists a sequence of rules that leads to this word when applied from the axiom S, i.e. $S \Rightarrow^* w$. A nonterminal A is reachable from the axiom if there exists $\alpha, \beta \in \Delta^*$ such that $S \Rightarrow^* \alpha A \beta$.

The language generated by the grammar is the set of derivable words from its axiom and is noted $L(G)$.

For a given sentential form $w = x_1 x_2 \ldots x_m$, where $m \in \mathbb{N}$ and $x_i \in \Delta$ for $i = 1, 2 \ldots m$, the reversal of w is the word $w^r = x_m x_{m-1} \ldots x_1$. We extend this notation to language: $L^r = \{w^r \mid w \in L\}$.

In the following, we will use the notion of quotient of languages, that can be defined as follows. If A and B are languages (i.e., subsets of T^*), the left quotient of A by B, written $B \diagdown A$, is the set $\{v \in T^* \mid \exists u \in T^* : uv \in A \land u \in B\}$. Similarly, the right quotient $A \diagup B$ is $\{u \in T^* \mid \exists v \in T^* : uv \in A \land v \in B\}$. We will denote $p \diagdown A$ the left quotient of A by the language containing solely the word $p \in T^*$ and $A \diagup s$ the right quotient of A by the language containing solely the word $s \in T^*$.

We will denote tuples in boldface. For example: $\mathbf{t} = (t_1, \ldots, t_m)$.

3.2. Grammar classes

The hierarchy of Noam Chomsky classifies grammars into several types [48], according to the restrictions on the form of the rules. This classification is briefly described in Table 1. A language L is said to be of type t whenever there exists a type-t grammar G such that $L = L(G)$. Many decision problems about formal languages depend on the language type [49, 50, 51]. In this work, we limit the scope of our research to injections in context-free languages, where the left-hand part of each rule consists of one non-terminal.

Type 0	Any grammar	no restriction on the rules form
Type 1	Context-sensitive grammar	rules of the form: $\alpha A\beta \to \alpha\gamma\beta$ with $A \in N, \alpha, \beta \in N^*, \gamma \in \Delta^+$
Type 2	Context-free grammar	rules of the form: $A \to \beta$ with $A \in N, \beta \in \Delta^*$
Type 3	Regular grammar	rules of the form: $A \to aB$, $A \to a$ or $A \to \epsilon$ with $A, B \in N$, $a \in T$

Table 1: Chomsky hierarchy

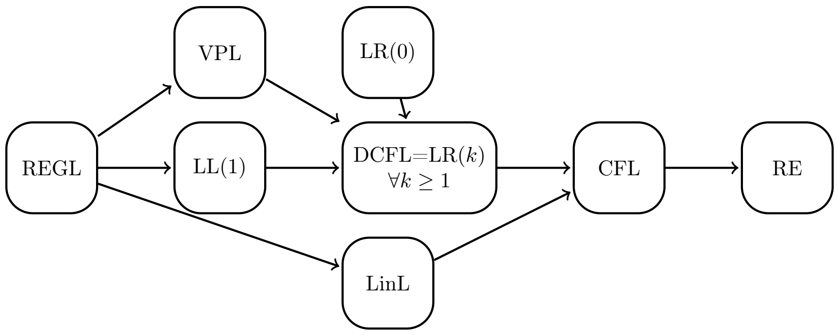


Figure 1: An arrow from class A to B indicates that A is a strict subset of B.

For context-free languages, one can represent a derivation in the form of a rooted tree, called *derivation tree*. Each node is labelled by an element of Δ; the root is labelled by the axiom S, each internal node is labelled by a nonterminal and each leaf is labelled by a terminal. A context-free grammar is said *ambiguous* if there exists at least one word with multiple derivation trees, and *unambiguous* if such a word does not exist.

In addition to these classic grammar types, we will be interested in some other classes: deterministic context-free grammars, LR(k), LR(0) and LL(1) grammars. Deterministic context-free grammars [52] are derived from deterministic pushdown automata. They form a proper subset of context-free grammars and are widely used because they can be parsed in linear time.

A subset of deterministic context-free grammars are LR(k) grammars [53] (LR stands for "left-to-right, rightmost derivation") for any $k \geq 1$. LR(k) grammar definition from [54] (Def 2.1) relies on the concept of rightmost derivation. A rightmost derivation is a derivation in which at any derivation step the rightmost nonterminal is rewritten. Let us denote $\Rightarrow_R$ the rightmost derivation and $\Rightarrow_R^*$ its reflexive transitive closure. Let us also denote ${}^{(k)}w$ the prefix of w with length $\min(|w|, k)$. The definition of an LR(k) grammar is as follow: let $G = (N, T, R, S)$ be a grammar such that $S \not\Rightarrow_R^+ S$ and let $w, w', x \in T^*$, $\gamma, \alpha, \alpha', \beta, \beta' \in \Delta^*$, $A, A' \in N$. G is said to be LR(k) if the following implication is true: if $S \Rightarrow_R^* \alpha A w \Rightarrow_R \alpha\beta w = \gamma w$, $S \Rightarrow_R^* \alpha' A' x \Rightarrow_R \alpha'\beta' x = \gamma w'$ and ${}^{(k)}w = {}^{(k)}w'$, then $A = A', \beta = \beta', |\alpha\beta| = |\alpha'\beta'|$. This property makes them effectively parsed by an LR parser [53]. All LR(k) for $k \geq 1$ describe the same set of languages: the deterministic context-free languages. A strict subset of LR(k) grammars is the set of LR(0) grammars: they can be parsed without any lookahead.

Visibly pushdown languages are a strict subset of deterministic languages with many closures properties and decidable problems [55]. They rely on a partition of the alphabet in three sets: calls, returns, and local symbols. Each set can only affect the stack of the automata in some way.

Finally, linear grammars [56] are grammars where the right-hand part of the rules can contain at most one non-terminal. They are context-free grammars but may be non-deterministic. For example, the set $\{ww^r \mid w \in T^*\}$ is not deterministic [50] but can be easily expressed with the linear grammar $(\{S\}, T, \{S \to tSt \mid t \in T\} \cup \{S \to \epsilon\}, S)$.

Name	Grammar class	Language class
Regular	REGG	REGL
Visibly pushdown	VPG	VPL
Deterministic context-free	DCFG	DCFL
Linear	LinG	LinL
Context-free	CFG	CFL

Table 2: Abbreviation of some languages and grammars classes

Operation	REGL	LL(1)	LR(0)	VPL	DCFL	LinL	CFL
Concatenation	✓	✗	✓	✓	✗	✗	✓
Union	✓	✗	✗	✓	✗	✓	✓
Complement	✓	✗	✗	✓	✓	✗	✗
Intersection with regular set	✓	✗	✗	✓	✓	✓	✓
Inverse homomorphism	✓	✗	✗	✗	✓	✓	✓

Table 3: Closure (symbol ✓) and non-closures (symbol ✗) of various languages classes with respect to classic operations.

Problem	REGL	LL(1)	LR(0)	VPL	DCFL	LinL	CFL
Membership ($w \in L$?)	D	D	D	D	D	D	D
Equivalence ($L_1 = L_2$?)	D	D	D	D	D	U	U
Inclusion ($L_1 \subseteq L_2$?)	D	U	U	D	U	U	U
Emptiness ($L = \varnothing$?)	D	D	D	D	D	D	D

Table 4: Decidability (D) and undecidability (U) of various problems on languages classes.

Since we will be most interested in grammars, we differentiate the set of grammars from the set of languages. See Table 2 for a list of abbreviations used in this article. A summary of the inclusion properties of these languages is presented in Figure 1.

Table 3 summarizes the closures and non-closures properties of operations for classic grammar classes [56, 50, 55]. These operations are classical operations on sets (union, complement, intersection) and strings (concatenation, also called product). Homomorphisms are letter-to-string functions (i.e., $T \to T^*$). They are extended to string-to-string functions in a morphic way: $h(\epsilon) = \epsilon$, $h(uv) = h(u)h(v)$ for all strings $u, v \in T^*$.

Table 4 summarizes the decidability of various problems for the same grammar classes [57, 58]: membership, equivalence, inclusion and emptiness.

4. Motivation for malicious injections modelization

As shown in the introduction, injections are ubiquitous and should not be restricted to SQL injection attacks. In many systems, user inputs are used to build queries for other services. For example, in a logging system, a network probe can send data to the logging server for storage: the log system usually concatenates such data with some context (probe ID, time, etc.) and writes the result of this concatenation to a file, for example in XML format. A Security Information and Event Management (SIEM) that parses and processes each query in that file to identify anomalies could suffer an injection-based attack from this log file (CVE-2020-5013). As another example, on a system with distributed authentication systems, authentication requests are generally made with the LDAP protocol. Such queries are built by concatenating user inputs (login and password) with the rest of the query written by the developer.

In this paper, we use the term query to denote the complete built data that includes the part written by the developer and the part that stems from the user input. Remark that this query is not necessarily directly transmitted to another system: in the logging system example, the query is an XML file. We use the term template to denote the set of strings the developer writes to construct their query by concatenation. The term blank denotes a place in a template in which the user input can be inserted by concatenation. A template can contain either one blank (for instance, a search form) or multiple blanks (for instance, an authentication form). The data injected by users in the template is called an injection. We find the term "user input" inappropriate because the injection may not come directly from a human user (as it can be the case in second-order injection attacks). Besides, the user input may be modified (typically escaped or truncated) before being included in the template. Remark that, with this definition, an injection is not necessarily malicious: legitimate injections exist as well and are expected.

For the sake of illustration, assume the developer maintains a search system in a product database. It allows selecting products in a given category whose price is below or above a threshold. They design the following template, which respects the previous semantics:

$$\textbf{SELECT * FROM product WHERE price } \llcorner\!\lrcorner_1 \textbf{ OR category } = \llcorner\!\lrcorner_2$$

This template has two blanks, denoted $\llcorner\!\lrcorner_1$ and $\llcorner\!\lrcorner_2$, where user inputs can be injected. A web page asks the user to provide the comparison operator with the threshold and the category to build the query. The develop expects the user to enter a comparison operator followed by a number in the first blank and a string in the second blank.

A malicious injection is an injection that modifies the semantics of the query. Formal language theory does not model semantics. However, the semantics is handled by the parser that heavily relies on a grammar. Even though there is an infinity of grammars that describe any language, there is generally one grammar of reference for every programming language. Such grammar has not been chosen at random: it is optimized to be understandable by the human (for example, the name of nonterminals generally conveys some semantics about its role in the grammar) and to facilitate the production of the semantics by the parser (generally used by a compiler or an interpreter). For these reasons, we make the reasonable assumption that the syntactic analysis by the grammar of reference of a language strongly correlates the sentence's semantics. Classic injection attacks clearly illustrate that they must indeed modify the order and the types of the tokens of the queries drastically to modify the way the sentence is parsed and thus its semantics.

When a developer writes a template, they have some intent about the user input: a name, a number, or a more complex structure such as a Boolean expression for a condition. The intent may be simple (for instance, a number or a string) or complex (for instance, a Boolean expression or a comparison operator followed by a variable name). Our fundamental assumption is that these intents correspond to symbols or sequences of symbols of the grammar of reference of that language. We consider that legitimate injections respect the intent while malicious injections don't. These terms will be formally defined in the next section.

In our database query example, the double injection ($\geq$ **120**, **"book"**) is legitimate as they respect the intents: the first one contains a comparison operator with a number and the second one contains a string. On the other hand, the double injection ($\geq$ **120 OR 1** $=$ **1**, **"book"**) is malicious, just like ($\geq$ **120**, **"book" OR 1** $=$ **1**). Indeed, these two injections result in syntactically correct queries, but they change the semantics of the query (they bypass the conditions and make the database return every product), and they do not correspond to the developer's initial intent.

Our study focuses on two new security properties: the intent-equivalence and the intent-security. This paragraph describes shortly these properties and how their study could be used to enhance security tools. The intent-equivalence is a property that a template verifies when it accepts no malicious injections. We discuss the decidability of this property for various grammars classes. This property is notably useful for static analysis to detect vulnerabilities in a system and our result could be used to enhance filtering and other user input processing. The intent-security is a property that a grammar verifies when every template one can write in that grammar is intent-equivalent. This property, stronger than the intent-equivalence, can be used to prove that a query language or a protocol is not vulnerable to injection-based attacks. Finally, we characterize the set of malicious injections of various grammar classes. This analysis is of interest for black-box fuzzing, whose objective is to test a system to find unexpected injections.

This article aims at providing results, answers, and advice for these various application cases.

5. Intent-equivalence

In this section, we define and study the intent-equivalence property.

5.1. Formal definition of the intent-equivalence

For the sake of simplicity, we will first restrict our definition to templates with a single blank. The definitions will be extended to templates with multiple blanks at the end of this subsection. In the single-blank setting, a template $p\_\_s$ can be described as a prefix p and a suffix s. In the following, every definition are given with respect to a grammar $G = (N, T, R, S)$.

Let us begin by defining the set of injections that can be used in some template. This definition is straightforward: it is the set of factors that lead to a valid (syntactically correct) word when inserted into the template.

Definition 5.1 (Injections of L for a template (p, s))**.** Let L be a formal language and $p, s \in T^*$. We define the language of injections of L associated to a template (p, s) as the language defined by:

$$F(L, (p, s)) = \{w \in T^* \mid pws \in L\} = p \diagdown L \diagup s$$

Among these injections, there may be legitimate and malicious ones. As proposed in the previous section, we formalize the legitimate injections by the set of words that can be derived from a sentential form we call the intent. If the intent of the developer is some $\iota \in \Delta^+$, then they expect a user input that can be derived from ι. Therefore, we call the expected injections for intent ι the injections that can be derived from ι.

Definition 5.2 (Expected injections of G for intent ι)**.** Let $\iota \in \Delta^+$. The language of expected injections of G for intent ι, noted $E(G, \iota)$, is defined as:

$$E(G, \iota) = \{w \in T^* \mid \iota \Rightarrow^* w\}$$

Notice that the set of expected injections of a regular (resp. context-free) grammar G is also a regular (resp. context-free) language. So this set is generally as easy to manipulate as G. So, given an intent, it is easy to verify whether or not some injection is compatible with it, i.e., if the injection is a member of the expected injections associated to that intent.

Remark that this definition of expected injections is adapted to context-free grammars but not to context-sensitive grammars. Indeed, in context-sensitive grammars, the strings consistent with an intent could depend on the context (i.e., the template). For this reason, we limit our study to context-free languages. We do not consider this limitation of our framework as very restrictive, since almost all programming languages have context-free grammars[5].

Proposition 5.3. *Let G be a context-free grammar, $\iota \in \Delta^+$ and $w \in T^*$. It is decidable whether or not $w \in E(G, \iota)$.*

Proof. Let $G' = (N \cup \{S'\}, T, R \cup \{S' \to \iota\}, S')$ where S' is a new nonterminal. This grammar is context-free and recognizes the language $L(G') = \{w \mid \iota \Rightarrow^*_G w\} = E(G, \iota)$. The membership problem for context-free grammar is decidable [59]. Thus, it is decidable if $w \in E(G, \iota)$. □

If ι is a sequence of terminals, the only sentential form w for which $\iota \Rightarrow^* w$ is ι itself and therefore $E(G, \iota) = \{\iota\}$. Such intent may seem useless because then the developer would expect only one possible injection, removing all information gained from the user input. However, in general, the terminals of real-world grammars are tokens whose recognition is handled by the lexer, and tokens like "string" or "number" entail multiple possibilities for user inputs. So, most intent are in fact terminals.

A template is secure if the set of all injections is explained by at least one intent ι that could appear in that sentence. In that case, we say that the template is intent-equivalent to ι.

Definition 5.4 (Intent-equivalence to an intent)**.** Let G be a formal grammar and $p, s \in T^*$ and $\iota \in \Delta^+$. The template (p, s) is intent-equivalent to ι if:

$$S \Rightarrow^* p\iota s \qquad \text{and} \qquad F(L(G), (p, s)) = E(G, \iota)$$

Remark 5.5. *To prove that (p, s) is intent-equivalent to ι, it suffices to show that $S \Rightarrow^* p\iota s$ and $F(L(G), (p, s)) \subseteq E(G, \iota)$, as $S \Rightarrow^* p\iota s$ implies $E(G, \iota) \subseteq F(L(G), (p, s))$.*

Remark 5.6. *$S \Rightarrow^* \alpha$ ($\alpha \in \Sigma^*$) is always decidable because G is context-free. Indeed, one can create a grammar $G' = (N, T \cup \{t_A \mid A \in N\}, R \cup \{A \to t_A \mid A \in N\}, S)$ that creates a new terminal for each non-terminal, so each sentential form of $L(G)$ can be mapped to a word of $L(G')$ with the following bijective homomorphism h:*

$$h(a) = \begin{cases} t_a & \text{if } a \in N \\ a & \text{if } a \in T \end{cases}$$

By construction, $S \Rightarrow^*_G \alpha$ if and only if $S \Rightarrow^*_{G'} h(\alpha)$, which is decidable because G' is context-free. In particular $S \Rightarrow^*_G p\iota s$ if and only if $S \Rightarrow^*_{G'} h(p\iota s)$.

In various cases, a template can include multiple blanks. A classic example is the template associated to a login page that has generally a blank for the username and another for the password. We generalize the previous definitions for templates with m blanks and intentions that are tuples. As a reminder, we denote tuples in boldface.

[5]See Antlr Project (`https://github.com/antlr/grammars-v4`)

A template is a sequence of terminals with some blanks: $t_1 \sqcup_1 t_2 \sqcup_2 t_3 \dots t_m \sqcup_m t_{m+1}$. We impose the presence of at least one terminal between two blanks; without this constraint, one blank could be interpreted as several adjacent blanks.

Definition 5.7 (Template with m blanks)**.** Let $m \geq 1$. A <u>template with m blanks</u> is a sequence $\mathbf{t} = (t_1, \dots, t_{m+1}) \in T^* \times (T^+)^{m-1} \times T^*$.

Given a template with m blanks and m injections, we define the fill-in-the-blanks operator that creates the concatenation of the template with the injections.

Definition 5.8 (Fill-in-the-blanks operator)**.** Let $m \geq 1$. We define the fill-in-the-blanks operator $\odot$ as:

$$\begin{aligned} \odot : \ & (T^* \times (T^+)^{m-1} \times T^*) \times (T^*)^m \to T^* \\ & (t_1, \dots, t_{m+1}), (w_1, \dots, w_m) \mapsto t_1 w_1 t_2 w_2 \dots w_m t_{m+1} \end{aligned}$$

We will use the infix notation (e.g. $\mathbf{t} \odot \mathbf{w}$, read "$\mathbf{t}$ filled with $\mathbf{w}$") for this operator.

Let us expand the previous definitions (5.1, 5.2, and 5.4) to templates with m blanks.

Definition 5.9 (Injections of L for a template $\mathbf{t}$)**.** Let L be a formal language, $m \geq 1$ and $\mathbf{t} \in T^* \times (T^+)^{m-1} \times T^*$. We define the language of <u>injections of L associated to a template $\mathbf{t}$</u> as the language defined by:

$$F(L, \mathbf{t}) = \{\mathbf{w} \in (T^*)^m \mid \mathbf{t} \odot \mathbf{w} \in L\}$$

Definition 5.10 (Expected injections of G for intents $\boldsymbol{\iota}$)**.** Let G be a formal grammar, $m \geq 1$ and $\boldsymbol{\iota} = (\iota_1, \iota_2, \dots, \iota_m) \in (\Delta^+)^m$. The language of <u>expected injections of G for intents $\boldsymbol{\iota}$</u>, noted $E(G, \boldsymbol{\iota})$, is defined as:

$$E(G, \boldsymbol{\iota}) = E(G, \iota_1) \times \dots \times E(G, \iota_m)$$

Definition 5.11 (Intent-equivalence of a template to an intent)**.** Let G be a formal grammar, $m \geq 1$, $\mathbf{t} \in T^* \times (T^+)^{m-1} \times T^*$ and $\boldsymbol{\iota} \in (\Delta^+)^m$. The template $\mathbf{t}$ is <u>intent-equivalent to $\boldsymbol{\iota}$</u> if:

$$S \Rightarrow^* \mathbf{t} \odot \boldsymbol{\iota} \qquad \text{and} \qquad F(L(G), \mathbf{t}) = E(G, \boldsymbol{\iota})$$

5.2. Decidability of intent-equivalence

In this subsection, we are interested in the decidability of the intent-equivalence of a template $\mathbf{t}$ with m blanks to some fixed intent $\boldsymbol{\iota}$ for various classes of grammar. This problem mainly concerns the static analysis of the source code, where we have access to the template. The first class we study is those of the regular grammars.

Theorem 5.12. *Let $G = (N, T, R, S)$ be a regular grammar, $m \geq 1$, $\mathbf{t}$ a template with m blanks and $\boldsymbol{\iota} \in (\Delta^+)^m$. It is decidable whether $\mathbf{t}$ is intent-equivalent to $\boldsymbol{\iota}$.*

Proof. To decide the intent-equivalence, we have to prove that $F(L, \mathbf{t}) = E(G, \boldsymbol{\iota})$. These two sets contain m-tuples and, therefore, are not languages. To apply results from language theory, we transform these tuples into words with the injective function $f_{\mathbf{t}}$ that relies on new symbols $\#_i, \overline{\#}_i$, for $m \geq i \geq 1$ and defined as follow:

$$f_{\mathbf{t}}(\mathbf{w}) = \mathbf{t} \odot (\#_1 w_1 \overline{\#}_1, \#_2 w_2 \overline{\#}_2, \dots, \#_m w_m \overline{\#}_m)$$

If $f_{\mathbf{t}}(F(L, \mathbf{t})) = \{f_{\mathbf{t}}(\mathbf{w}) \mid \mathbf{w} \in F(L, \mathbf{t})\}$ is equal to $f_{\mathbf{t}}(E(G, \boldsymbol{\iota})) = \{f_{\mathbf{t}}(\mathbf{w}) \mid \mathbf{w} \in E(G, \boldsymbol{\iota})\}$ then $F(L, \mathbf{t}) = E(G, \boldsymbol{\iota})$ due to $f_{\mathbf{t}}$ being injective.

Let's first construct the set $f_{\mathbf{t}}(F(L, \mathbf{t}))$ with the languages L_1 and L_2. L_1 is the language L whose each word have been doped with the new symbols $\#_i, \overline{\#}_i$ for $m \geq i \geq 1$. Consider the following homomorphism h:

$$h(a) = \begin{cases} \epsilon & \text{if } a = \#_i \text{ or } a = \overline{\#}_i \text{ for some } m \geq i \geq 1 \\ a & \text{if } a \in T \end{cases}$$

This homomorphism removes the symbol $\#_i$ and $\overline{\#}_i$, so its inverse adds these symbols and $L_1 = h^{-1}(L)$. Since regular languages are closed under inverse homomorphisms, L_1 is regular. Remark that $f_{\mathbf{t}}(\mathbf{w}) \in L_1$ if and only if $\mathbf{t} \odot \mathbf{w} \in L$.

L_2 is the set of templates completed with any factors where each factor is surrounded by $\#_i$ and $\overline{\#}_i$. Remark that, with a slight abuse of notation, $L_2 = \mathbf{t} \odot (\#_1 T^* \overline{\#}_1, \#_2 T^* \overline{\#}_2, \dots, \#_m T^* \overline{\#}_m)$. As a finite concatenation of regular languages, this language is regular. Furthermore, it does not depend on the language L. We will now prove that $f_{\mathbf{t}}(F(L, \mathbf{t})) = L_1 \cap L_2$.

$\subseteq$: Let $\mathbf{w} \in F(L, \mathbf{t})$. Remark that $L_2 = \{f_{\mathbf{t}}(\mathbf{w}) \mid \mathbf{w} \in (T^*)^m\}$ so $f_{\mathbf{t}}(\mathbf{w}) \in L_2$. Besides, $\mathbf{t} \odot \mathbf{w} \in L$, so $f_{\mathbf{t}}(\mathbf{w}) \in L_1$. So $f_{\mathbf{t}}(F(L, \mathbf{t})) \subseteq L_1 \cap L_2$.

$\supseteq$: Let $w \in L_1 \cap L_2$. Since $L_2 = \{f_{\mathbf{t}}(\mathbf{w}) \mid \mathbf{w} \in (T^*)^m\}$, there exists $\mathbf{w}' \in (T^*)^m$ such that $w = f_{\mathbf{t}}(\mathbf{w}')$. Since $f_{\mathbf{t}}(\mathbf{w}') \in L_1$, we know that $\mathbf{t} \odot \mathbf{w}' \in L$, so $\mathbf{w}' \in F(L, \mathbf{t})$. So $L_1 \cap L_2 \subseteq f_{\mathbf{t}}(F(L, \mathbf{t}))$.

Finally, $f_{\mathbf{t}}(F(L, \mathbf{t})) = L_1 \cap L_2$. Since L_1 and L_2 are regular, the set $f_{\mathbf{t}}(F(L, \mathbf{t}))$ is regular. Besides, $f_{\mathbf{t}}(E(G, \boldsymbol{\iota})) = \{\mathbf{t} \odot (\#_1 w_1 \overline{\#}_1, \#_2 w_2 \overline{\#}_2, \dots, \#_m w_m \overline{\#}_m) \mid \forall i, w_i \in E(G, \iota_i)\}$. Since every language $E(G, \iota_i)$ is regular, $f_{\mathbf{t}}(E(G, \boldsymbol{\iota}))$ is regular.

So the template $\mathbf{t}$ is intent-equivalent to $\boldsymbol{\iota}$ if and only if $f_{\mathbf{t}}(F(L, \mathbf{t})) = f_{\mathbf{t}}(E(G, \boldsymbol{\iota}))$. Since both languages are regular, this is decidable (by Theorem 3.12 from [50]). □

This result should not be surprising as most problems are decidable for regular grammars. To expand this result to deterministic language, we would need to compare the languages of expected injections $E(G, \boldsymbol{\iota})$ with the language of injections. However, this may not be easy. Consider the case with one blank, where $\mathbf{t} = (p, s)$. In that case, $F(L, (p, s)) = p \diagdown L(G) \diagup s$. But while $p \diagdown L(G) \diagup s$ is necessarily deterministic, we don't know whether $E(G, \iota)$ is deterministic or not in the general case. Nonetheless, we prove in the following lemma that for an intent of length 1 ($\iota \in \Delta$), if G is an LR(k) grammar then $E(G, \iota)$ is deterministic.

Lemma 5.13. *Let $k \in \mathbb{N}$ and $G = (N, T, R, S)$ be an LR(k) grammar. Then, for any $A \in \Delta$ reachable from S, $G' = (N, T, R, A)$ is a LR(k) grammar.*

Proof. This proof relies on the definition of LR(k) grammars described in Section 3.2.

The case where A is a terminal is trivial, as in this case $E(G, A) = \{A\}$ is finite. In the following, we assume that A is a nonterminal. Since A is reachable from S, there exist $\delta \in \Delta^*$ and $w \in T^*$ such that $S \Rightarrow_R^* \delta A w$. Let $\alpha, \gamma \in \Delta^*$, $w_1, w_2, w_3 \in T^*$ such that:

$$\begin{aligned} &A \Rightarrow_R^* \alpha B w_2 \Rightarrow_R \alpha\beta w_2 \\ &A \Rightarrow_R^* \gamma C w_1 \Rightarrow_R \alpha\beta w_3 \\ &{}^{(k)}w_2 = {}^{(k)}w_3 \end{aligned}$$

G' is LR(k) if $\alpha = \gamma$, $B = C$ and $w_1 = w_3$. Since $S \Rightarrow_R^* \delta A w$:

$$S \Rightarrow_R^* \delta A w \Rightarrow_R^* \delta\alpha B w_2 w \Rightarrow_R \delta\alpha\beta w_2 w$$
$$S \Rightarrow_R^* \delta A w \Rightarrow_R^* \delta\gamma C w_1 w \Rightarrow_R \delta\alpha\beta w_3 w$$

Remark that if ${}^{(k)}w_2 = {}^{(k)}w_3$ then ${}^{(k)}(w_2 w) = {}^{(k)}(w_3 w)$. We can use our hypothesis that G is LR(k) and conclude that $\delta\alpha = \delta\gamma$, $B = C$, $w_1 w = w_3 w$. Hence $\alpha = \gamma$ and $w_1 = w_3$, so G' is $LR(k)$. □

This lemma allows us to adapt the proof of Theorem 5.12 and to conclude the decidability of the intent-equivalence for LR(k) grammar when the intent length is 1.

Theorem 5.14. *Let G be an LR(k) grammar, $m \geq 1$, $\mathbf{t}$ a template with m blanks and $\boldsymbol{\iota} \in (\Delta)^m$ (so $|\iota_i| = 1$ for all i). It is decidable whether $\mathbf{t}$ is intent-equivalent to $\boldsymbol{\iota}$.*

Proof. The proof is similar to the proof of Theorem 5.12. Deterministic languages are closed by inverse homomorphism, so L_1 is deterministic. They are also closed by intersection with a regular language, so $L_1 \cap L_2$ is deterministic (L_2 doesn't depend on L and is always regular). Because each $E(G, \iota_i)$ is LR(k) (due to Lemma 5.13) and because $f_{\mathbf{t}}(F(L,t))$ is a marked concatenation of LR(k) languages, $f_{\mathbf{t}}(F(L,t))$ is deterministic [54]. So $F(L, \mathbf{t})$ and $E(G, \boldsymbol{\iota})$ are deterministic. Finally, the equivalence of deterministic languages is decidable [60]. We can conclude that it is decidable whether $\mathbf{t}$ is intent-equivalent to $\boldsymbol{\iota}$. □

This theorem is limited to $|\iota_i| = 1$ (i.e., to intent length equal to 1) because, otherwise, $E(G, \iota_i)$ may not be LR(k). However, LR(0) grammars are closed by concatenation, so we obtain the following result:

Theorem 5.15. *Let G be an LR(0) grammar, $m \geq 1$, $\mathbf{t}$ a template with m blanks and $\boldsymbol{\iota} \in (\Delta^+)^m$ (no constraint on the size of ι_i). It is decidable whether $\mathbf{t}$ is intent-equivalent to $\boldsymbol{\iota}$.*

Proof. The proof is similar to the proof of 5.14. Denote $\iota_i = \iota_{i,1}\iota_{i,2}\ldots\iota_{i,k}$. We can decompose $E(G, \iota_i)$ in the following way: $E(G, \iota_i) = E(G, \iota_{i,1}) \cdot E(G, \iota_{i,2}) \cdot \ldots \cdot E(G, \iota_{i,k})$. Due to previous lemma, each set $E(G, \iota_{i,j})$ (for $1 \leq j \leq k$) is an $LR(0)$ language. Since the languages $LR(0)$ are closed by product [54], $E(G, \iota_i)$ is an $LR(0)$ language. The rest of the proof is similar. □

The same result can be obtained for visibly pushdown grammars, for about the same reasons.

Theorem 5.16. *Let G be an visibly pushdown grammar, $m \geq 1$, $\mathbf{t}$ a template with m blanks and $\boldsymbol{\iota} \in (\Delta^+)^m$ (no constraint on the size of ι_i). It is decidable whether $\mathbf{t}$ is intent-equivalent to $\boldsymbol{\iota}$.*

Proof. Let $G = (N, T, R, S)$ a visibly pushdown grammar. The proof is similar to the previous one. Denote $\iota_i = \iota_{i,1}\iota_{i,2}\ldots\iota_{i,k}$. We can decompose $E(G, \iota_i)$ in the following way: $E(G, \iota_i) = E(G, \iota_{i,1}) \cdot E(G, \iota_{i,2}) \cdot \ldots \cdot E(G, \iota_{i,k})$. Each set $E(G, \iota_{i,j})$ is either described by the grammar $G = (N, T, R, \iota_{i,j})$ if $\iota_{i,j}$ is a nonterminal, or is finite if $\iota_{i,j}$ is a terminal. In both case, $E(G, \iota_{i,j})$ is visibly pushdown. Since the visibly pushdown languages are closed by product [55], $E(G, \iota_i)$ is a visibly pushdown language. As a concatenation of visibly pushdown languages, $E(G, \boldsymbol{\iota})$ is visibly pushdown. The rest of the proof is similar. □

These results could not be extended directly to LL(k) grammars (not even LL(1) grammars) as they are not closed by concatenation [58].

The previous theorems are limited to deterministic grammars. It would be interesting to see whether the intent-equivalence could be decidable for more general context-free grammars. However, the following theorem shows that this is undecidable even for a single blank and for a subset of context-free grammars.

Theorem 5.17. *Let G be a linear grammar, (p, s) a template and $\iota \in \Delta$. It is undecidable whether (p, s) is intent-equivalent to ι.*

Proof. Let $G_1 = (N_1, T, R_1, S_1)$ and $G_2 = (N_2, T, R_2, S_2)$ two linear grammars. Let S_3, S_2', $\#_1$, $\#_2$ be new symbols and $G_3 = (T_3 = T \cup \{\#_1, \#_2\}, N_3 = N_1 \cup N_2 \cup \{S_2', S_3\}, R_3 = R_1 \cup R_2 \cup R', S_3)$ where R' is defined as:

$$S_3 \to \#_1 S_1 \qquad S_3 \to \#_1 S_2' \qquad S_2' \to S_2 \qquad S_2' \to \#_2$$

Due to the rules in R', $F(L(G_3), (\#_1, \epsilon)) = L(G_1) \cup L(G_2) \cup \{\#_2\}$. Let search what symbol $\iota \in T_3 \cup N_3$ can explain these injections. We can break this search into four cases:

1. $\iota \neq S_3$, $\iota \neq S_2'$, $\iota \neq \#_2$. Then $E(G_3, \iota)$ does not contain the symbol $\#_2$ so $F(L_3, (\#_1, \epsilon)) \neq E(G_3, \iota)$.
2. $\iota = \#_2$. Then $E(G_3, \iota) = \{\#_2\}$ and $F(L(G_3), (\#_1, \epsilon)) \neq E(G_3, \iota)$ iff $L(G_1) = L(G_2) = \varnothing$.
3. $\iota = S_3$. But $S_3 \not\Rightarrow^*_{G_3} \#_1 S_3$, so $(\#_1, \epsilon)$ cannot be intent-equivalent to S_3.
4. $\iota = S_2'$. $S_3 \Rightarrow^*_{G_3} \#_1 S_2'$ so it is a suitable candidate and $E(G, S_2') = L(G_2) \cup \{\#_2\}$.

Since $(\#_1, \epsilon)$ might only be intent-equivalent to S_2' or $\#_2$, we deduce that:

$$\begin{aligned}
&(\#_1, \epsilon) \text{ is intent-equivalent to some } \iota \\
&\iff F(L(G_3), (\#_1, \epsilon)) \neq E(G_3, \iota) \vee F(L(G_3), (\#_1, \epsilon)) \neq E(G_3, \iota) \\
&\iff (L(G_1) \cup L(G_2) \cup \{\#_2\}) = (L(G_2) \cup \{\#_2\}) \vee L(G_1) = L(G_2) = \varnothing \\
&\iff L(G_1) - L(G_2) = \varnothing \\
&\iff L(G_1) \subseteq L(G_2)
\end{aligned}$$

The grammar G_3 is linear (due to the linearity of G_1 and G_2 and the form of the new rules), so if the intent-equivalence for linear grammars were decidable, then $L(G_1) \subseteq L(G_2)$ would be decidable. Because the latter is undecidable [61, 62] we conclude that the intent-equivalence is undecidable for linear grammars. □

Corollary 5.18. *Let G be a context-free grammar, (p, s) a template and $\iota \in \Delta$. It is undecidable whether (p, s) is intent-equivalent to ι.*

Proof. Context-free grammars are a superset of linear grammars. □

The origin of this undecidability is that the set of expected injections can be very complex. If we impose the intents to be composed of terminals only, the set of expected injections will be finite. In that case, the intent-decidability is decidable for context-free languages.

Theorem 5.19. *Let G be a context-free grammar, $m \geq 1$, $\mathbf{t}$ a template with m blanks and $\boldsymbol{\iota} \in (T^+)^m$, so ι can only contain terminals. It is decidable whether $\mathbf{t}$ is intent-equivalent to $\boldsymbol{\iota}$.*

Proof. Once again, the proof is similar to the proof of Theorem 5.12. Context-free languages are closed by inverse homomorphism, so L_1 is context-free. They are also closed by intersection with a regular language, so $L_1 \cap L_2$ is context-free. Because each $E(G, \iota_i) = \{\iota_i\}$ (due to $\iota_i \in T^+$) and because $f_t(F(L, t))$ is a concatenation of context-free languages, $f_t(F(L, t))$ is context-free. So $F(L, \mathbf{t})$ is context-free and $E(G, \boldsymbol{\iota})$ is finite, therefore $F(L, \mathbf{t}) - E(G, \boldsymbol{\iota})$ is context-free and its emptiness is decidable. We can conclude that it is decidable whether $\mathbf{t}$ is intent-equivalent to $\boldsymbol{\iota}$. □

This last result is interesting because the intents of the developer are generally composed of one token, i.e., one terminal.

Theorems 5.12, 5.14 and 5.15 indicate that security solutions though template analysis (e.g., by static analysis of the source code) can be useful as long as the developer's intent is known and the grammar is regular or LR(0). When the developer's intent length is 1, such analysis can also be performed on LR(k) grammars. When it is composed of terminals only, it is even possible on any context-free grammar. In the general case, however, even when the template and the developer's intent are known, it is undecidable whether unexpected injections can happen in linear or context-free grammars due to Theorem 5.18. The question of the decidability of the intent-equivalence for any intent is still open for deterministic context-free grammars.

The definitions of this section allow us to show whether some template is safe or not. However, this definition cannot be used to prove that a whole grammar is safe when the language is infinite. This is the subject of the next section.

6. Intent-security

6.1. Formal definition of the intent-security

The goal of this subsection is to propose a property that guarantees that a grammar is safe. Consider the following case, where a developer has some intent in mind. For example, they expect the user to provide an attribute name in a SQL query. Can they be guaranteed that any template written for that intent is intent-equivalent to their intent?

Let us first define the set of injections for a given intent for all templates. This set contains all words that could replace an expected injection among all templates. In the following, every definition are given with respect to a grammar $G = (N, T, R, S)$.

Definition 6.1. Let $\iota \in \Delta^+$ be a sentential form. The language of injections of G for intent ι, written $I(G, \iota)$, is the language defined by:

$$I(G, \iota) = \{w \in T^* \mid \exists p, s \in T^* : pws \in L(G), S \Rightarrow^* p\iota s\}$$

In all these injections, some may comply with the intent ι and be expected (and legitimate), and some may be unexpected.

Definition 6.2 (Unexpected injections of G for intent ι)**.** Let $\iota \in \Delta^+$. The language of unexpected injections of G for intent ι, noted $\delta I(G, \iota)$, is defined as:

$$\delta I(G, \iota) = \{w \in T^* \mid \exists p, s \in T^* : pws \in L(G), S \Rightarrow^* p\iota s, \iota \not\Rightarrow^* w\}$$

Remark 6.3. *By relying on expected injections, we propose an alternative definition of unexpected injections:*

Let $\iota \in \Delta^+$. The language of unexpected injections of G for intent ι can be defined as:

$$\delta I(G, \iota) = I(G, \iota) - E(G, \iota)$$

The two definitions are equivalent:

$$\begin{aligned}\delta I(G, \iota) &= \{w \in T^* \mid \exists p, s \in T^* : pws \in L(G) \text{ and } S \Rightarrow^* p\iota s \text{ and } \iota \not\Rightarrow^* w\} \\ &= \{w \in T^* \mid \exists p, s \in T^* : pws \in L(G) \text{ and } S \Rightarrow^* p\iota s\} - \{w \in T^* \mid \iota \Rightarrow^* w\} \\ &= I(G, \iota) - E(G, \iota)\end{aligned}$$

We can use this set of unexpected injections of G for intent ι to define properly the property of intent-security we describe earlier: an intent-secure grammar has no unexpected injections.

Definition 6.4 (Intent-secure grammar for an intent)**.** Let $\iota \in \Delta^+$. A formal grammar G is said to be intent-secure for ι if there are no unexpected injections for that intent, i.e.:

$$\delta I(G, \iota) = \varnothing$$

Since the intent-security holds for every prefix and suffix, it intuitively means that it is stronger than intent-equivalence. This is confirmed by the next proposition:

Proposition 6.5. *Let $\iota \in \Delta^+$. G is intent-secure for ι if and only if, for all templates (p, s) such that $S \Rightarrow^* p\iota s$, (p, s) is intent-equivalent to ι.*

Proof.

$$\begin{aligned} G \text{ is intent-secure for } \iota &\iff \delta I(G, \iota) = \varnothing \\ &\iff I(G, \iota) \subseteq E(G, \iota) \\ &\iff \iota \Rightarrow^* w \text{ for all } w, p, s \in T^* \text{ such that } pws \in L(G) \text{ and } S \Rightarrow^* p\iota s \\ &\iff (p, s) \text{ is intent-equivalent to } \iota \text{ for all } (p, s) \in T^* \text{ such that } S \Rightarrow^* p\iota s \end{aligned}$$ □

This result highlights the potential of grammar analysis. If one can prove that a grammar is intent-secure for some intent, then it means that a developer with that intent will always write templates that allow only legitimate injections.

Furthermore, we define the set of unexpected injections of G over all intents with length n:

Definition 6.6 (Unexpected injections of G for all intent of length n)**.** Let $n \geq 1$. The set of unexpected injections of G over all intents with length n, noted $\delta\mathcal{I}^n(G)$, is defined as:

$$\delta\mathcal{I}^n(G) = \{\delta I(G, \iota) \mid \iota \in \Delta^n\}$$

Remark that $\delta\mathcal{I}^n(G)$ is a set of languages and not a language. For this reason, we use the calligraphic font $\mathcal{I}$ and not I.

We could define the intent-security of a grammar as the property of being intent-secure for all possible intent. However, with that naive definition, nearly all grammars would be intent-insecure, as indicated by the following result.

Proposition 6.7. *Let G be a grammar such that $|L(G)| \geq 2$ and let l be the minimal length of the words of $L(G)$. Then G is not intent-secure over all intents with length l.*

Proof. Let w_1 be a word of smallest length in $L(G)$ and $w_2 \in L(G)$ such that $w_1 \neq w_2$. Remark that $S \Rightarrow^* w_1, S \Rightarrow^* w_2$ and $w_1 \not\Rightarrow^* w_2$. So $w_2 \in \delta I(G, w_1)$. □

So a grammar of practical interest cannot be intent-secure for any intent length. For this reason, we will define the intent-security of a grammar for some length of intent.

Definition 6.8 (Intent-secure grammar for intent length n)**.** Let $n \geq 1$. A formal grammar G is said to be intent-secure for intent length n if, for every intent of length equal to n, all injections are expected, i.e.:

$$\delta\mathcal{I}^n(G) = \{\varnothing\}$$

That means that a grammar can be intent-secure for simple intents whose length are 1, but not intent-secure for more complex intents. Let us illustrate this property with an infinite language that is intent-secure for intent length 1.

Proposition 6.9. *There exist context-free grammars generating infinite languages that are intent-secure for intent length 1.*

Proof. Let us consider the grammar $G = (\{a, b, c, d\}, \{S\}, R, S)$ where $R = \{S \to aSb; S \to cd\}$. A simple proof by induction shows that $L(G) = \{a^n cdb^n \mid n \geq 0\}$. Each word in $L(G)$ contains exactly one occurrence of the symbols c and d and contains as many a symbols as b symbols. All a symbols (resp. b symbols) of a word in $L(G)$ are before the symbol c (resp. after the symbol d).

The grammar G contains one non-terminal S and four terminals $\{a, b, c, d\}$. So there are only five different intents of length 1:

- Case 1: $\iota = a$. Due to the form of the words in $L(G)$, a query with a injection in a has the form $a^m \iota a^r cdb^n$, with $m, n, r \geq 0$. Since the word $a^m aa^r cdb^n$ belongs to $L(G)$, $m+1+r = n$, so $r = n-1-m$ and $a^m \iota a^{n-1-m} cdb^n$. The query already contains the symbols c and d. Therefore, a valid injection cannot contain these symbols. The symbol ι is to the left of the c symbol. Therefore, a valid injection cannot contain a b symbol and a valid injection contains only a symbols. In order to balance the number of a symbols and b symbols, the injection must contain only one a symbol. Finally, the only valid injection is a and $\delta I(G, a) = \{a\} - \{a\} = \varnothing$.
- Case 2: $\iota = c$. A valid injection must include exactly one c and no d because there must be exactly one c and one d in the query. There cannot be any b in this injection because all b are after d. If this injection contains one or more a, there will be strictly more a than b in the query. So, finally, the only valid injection is c.
- Case 3: $\iota = d$. Similar to case 2.
- Case 4: $\iota = b$. Similar to case 1.
- Case 5: $\iota = S$. A simple induction shows that all the sentential forms with this injection point have the following form: $a^n cdb^n$, with $n \geq 0$. The set of injections is therefore $I(G, S) = \bigcup_{n \geq 0} a^n \diagdown L \diagup b^n = \{a^m cdb^m \mid m \geq 0\} = E(G, S)$. So $\delta I(G, S) = \varnothing$.

Finally, $\delta \mathcal{I}(G) = \{\varnothing\}$. □

We succinctly extend the previous definitions to injections in multiple-blanks templates.

Definition 6.10 (Injections of G for intents $\boldsymbol{\iota}$)**.** Let $m \geq 1$ and $\boldsymbol{\iota} \in (\Delta^+)^m$. We define the language of injections of G at $\boldsymbol{\iota}$ as the language defined by:

$$I(G, \boldsymbol{\iota}) = \{\mathbf{w} \in (T^*)^m \mid \exists \mathbf{t} \in T^* \times (T^+)^{m-1} \times T^* : \mathbf{t} \odot \mathbf{w} \in L(G) \text{ and } S \Rightarrow^* \mathbf{t} \odot \boldsymbol{\iota}\}$$

Definition 6.11 (Unexpected injections of G for intents $\boldsymbol{\iota}$)**.** Let $m \geq 1$ and $\boldsymbol{\iota} \in (\Delta^+)^m$. We define the language of unexpected injections of G at $\boldsymbol{\iota}$ as the language defined by:

$$\delta I(G, \boldsymbol{\iota}) = I(G, \boldsymbol{\iota}) - E(G, \boldsymbol{\iota})$$

Because intents may have different lengths for templates with multiple blanks, we define the intent-security with a constraint on the maximum length of each element of the intent.

Definition 6.12 (Intent-secure grammar with multiple blanks)**.** A formal grammar G is said to be intent secure with m blanks for intent length at most n if, for all intent $\boldsymbol{\iota} \in (\bigcup_{k=1}^{n} \Delta^k)^m$, $\delta I(G, \boldsymbol{\iota}) = \varnothing$.

As we will see later, a grammar can be intent-secure for one blank but not intent-secure for multiple blanks.

6.2. Inherently intent-secure and inherently intent-insecure languages

In the previous subsection, we defined the notion of intent-security for grammars only, for the reasons detailed in Section 4. However, the intuition tells us that some languages are not intent-secure, no matter what grammar is used. So, in this subsection, we define the notion of inherently intent-secure and inherently intent-insecure languages.

Definition 6.13 (Inherently intent-secure language for intent length n)**.** Let L be a formal language. L is inherently intent-secure on intent length n if all context-free grammars G such that $L(G) = L$ are intent-secure on intent length n.

Even though the notion of inherently intent-secure languages may appear attractive, the following theorem shows that it is not a property that useful languages can verify.

Proposition 6.14. *Inherently intent-secure languages for intent length 1 contain at most one word.*

Proof. Let $G = (N, T, R, S)$ be a grammar that describes a language L with a least two words, w_1 and w_2. Let us show that we can create G' such that G' is not intent-secure and $L(G') = L$. Denote $\alpha_1, \alpha_2, \dots$ and $\alpha'_1, \alpha'_2, \dots$ such that $S \Rightarrow_L \alpha_1 \Rightarrow_L \alpha_2 \Rightarrow_L \dots \Rightarrow_L w_1$ is a left derivation of w_1 and $S \Rightarrow_L \alpha'_1 \Rightarrow_L \alpha'_2 \Rightarrow_L \dots \Rightarrow_L w_2$ is a left derivation of w_2. Let k such that $\alpha_k = \alpha'_k$ and $\alpha_{k+1} \neq \alpha'_{k+1}$ (k exists since the starts of the derivations are the same and the ends are different). Denote A the leftmost nonterminal of α_k that is derived and denote $A \to \beta$ the production rule used in the left derivation of w_1 and $A \to \gamma$ the production rule used in the left derivation of w_2. Let us compare the languages $L_\beta = \{w \mid \beta \Rightarrow^* w\}$ and $L_\gamma = \{w \mid \gamma \Rightarrow^* w\}$. If $L_\beta = L_\gamma$, we can remove the rule $A \to \gamma$ from R and it would still describe the same language. We could therefore use the same reasoning with this new grammar to identify β' and γ'. Since the number of production rules is finite, at some point we will identify some β'' and γ'' such that $L_{\beta''} \neq L_{\gamma''}$. So, let's assume that $L_\beta \neq L_\gamma$ and denote (without loss of generality) w such that $w \in L_\beta$ and $w \notin L_\gamma$, i.e., $\beta \Rightarrow^* w$ and $\gamma \not\Rightarrow^* w$.

Let construct $G' = (N \cup \{B\}, T, R', S)$ where B is a new nonterminal and R' is defined as $(R \setminus \{A \to \gamma\}) \cup \{A \to B, B \to \gamma\}$. Remark that $L(G') = L(G)$. Let pAs a sentential form that can be derived from S in G. So $S \Rightarrow^*_{G'} pBs$. Furthermore, $S \Rightarrow^*_{G'} pAs \Rightarrow p\beta s \Rightarrow^*_{G'} pws$. However, since $\gamma \not\Rightarrow^* w$, we can conclude that $B \not\Rightarrow^* w$ and therefore that $w \in \delta I(G', B)$. So L is not inherently intent-secure. □

We similarly define inherently intent-insecure languages.

Definition 6.15 (Inherently intent-insecure language for intent length n)**.** Let L be a formal language. L is inherently intent-insecure for intent length n if all context-free grammars G such that $L(G) = L$ are intent-insecure for intent length n.

In a similar fashion, a language is intent-insecure for m blanks if all its context-free grammars are intent-insecure for m blanks.

The next proposition shows a sufficient condition for a language to be inherently insecure.

Proposition 6.16. *Let L be a language, $n \geq 1$. If there exist $p, s \in T^*$, $z \in T^n$ and $w \in T^*$ such that $z \neq w$, $pzs \in L$ and $pws \in L$, then L is inherently intent-insecure for n.*

Proof. Let G such that $L(G) = L$. Then $w \in \delta I(G, z)$ so $\delta I(G, z) \neq \varnothing$. Since this is true for any grammar G, we can conclude that L is inherently intent-insecure. □

It is still an open question of whether an inherently intent-insecure language for intent length 1 can be intent-secure for all intents consisting of a unique terminal.

This result can be applied to easily show that a language is inherently intent-insecure with a single counterexample. This is applied to popular programming languages in the next subsection.

6.3. Monotonicity properties

Let us exhibit some properties about the monotonic behavior of unexpected injections. Indeed, usual advice for developers is to use the least powerful language suitable for one task [43] with the intuition that simpler languages are less prone to injection vulnerabilities. In this subsection, we explore such kinds of properties and whether they hold in our framework.

Without any surprises, we can prove a monotonic property of unexpected injections for unambiguous grammars and, therefore, for deterministic grammars.

Theorem 6.17. *Let two unambiguous grammars $G_1 = (N, T, R_1, S)$ and $G_2 = (N, T, R_2, S)$ such that $R_1 \subseteq R_2$. Then $\delta I(G_1, \iota) \subseteq \delta I(G_2, \iota)$ for all $\iota \in \Delta^+$.*

Proof. Let $\iota \in \Delta^+$, $w \in T^*$ such that $w \in \delta I(G_1, \iota)$. Let us show that $w \in \delta I(G_2, \iota)$. Given that $w \in \delta I(G_1, \iota)$, we know that there exist $p, s \in T^*$ such that $pws \in L(G_1)$, $S \Rightarrow^*_{G_1} p\iota s$ and $\iota \not\Rightarrow^*_{G_1} w$. Since $R_1 \subseteq R_2$, $\alpha \Rightarrow^*_{G_1} \beta$ implies that $\alpha \Rightarrow^*_{G_2} \beta$ because all the rules used to derive β from α in G_1 can be used in G_2. Therefore, $pws \in L(G_2)$, $S \Rightarrow^*_{G_2} p\iota s$.

Let us prove that $\iota \not\Rightarrow^*_{G_2} w$ by contradiction, by assuming that $\iota \Rightarrow^*_{G_2} w$ and showing that G_2 cannot be unambiguous. Since $\iota \not\Rightarrow^*_{G_1} w$, it means that at least one rule in R_2 but not in R_1 is used to derive w from ι in G_2. So, there is a derivation path from S to pws that uses at least one rule from $R_2 - R_1$. On the other hand, since $S \Rightarrow^*_{G_1} pws$, that there is another derivation path from S to pws that only use rules from R_1. We created two different derivation paths from S to pws in G_2, which is in contradiction with it being unambiguous. Therefore, we can conclude that the assumption $\iota \Rightarrow^*_{G_2} w$ is false.

Since we proved that $pws \in L(G_2)$, $S \Rightarrow^*_{G_2} p\iota s$ and $\iota \not\Rightarrow^*_{G_2} w$, we conclude that $w \in \delta I(G_2, \iota)$ and finally that $\delta I(G_1, \iota) \subseteq \delta I(G_2, \iota)$ for any $\iota \in \Delta^+$. □

This result motivates the advice mentioned above as it applies to the widely used unambiguous grammars. However, such monotonic property is not true in the general case, as shown with the following counterexample.

Remark 6.18. *There exist two context-free grammars $G_1 = (N, T, R_1, S)$ and $G_2 = (N, T, R_2, S)$ such that $R_1 \subseteq R_2$ and $\delta I(G_1, \iota) \not\subseteq \delta I(G_2, \iota)$ for some $\iota \in N$.*

Let $T = \{a, b, c\}$, $N = \{S, M\}$, $R_1 = \{S \to aaM, S \to aabb, M \to cc\}$ and $R_2 = \{S \to aaM, S \to aabb, M \to cc, M \to bb\}$. Thus, $R_1 \subset R_2$ and $L(G_1) = L(G_2) = \{aabb, aacc\}$. $E(G_2, M) = \{bb, cc\}$.

The terminals go in pairs. For a blank corresponding to a terminal intent, the only possible value is entirely determined by one of the surrounding terminals. Thus, for any terminal $\iota \in T$, $\delta I(G_1, \iota) = \varnothing$ and $\delta I(G_2, \iota) = \varnothing$. For the axiom, we necessarily have $\delta I(G_1, S) = \delta I(G_2, S) = \varnothing$. The only sentential form containing M is aaM, for both grammars. Thus, $I(G_1, M) = \{bb, cc\}$. Since $E(G_1, M) = \{cc\}$, we conclude that $\delta I(G_1, M) = \{bb\}$. On the other hand, $E(G_2, M) = \{bb, cc\}$ so $\delta I(G_2, M) = \varnothing$.

We can also derive a monotonic property on languages (and not grammars), provided we restrict the analysis to terminal intents.

Proposition 6.19. *Let G_1 and G_2 be two grammars such that $L(G_1) \subseteq L(G_2)$. Let $n \geq 1$, $\iota \in T^n$. Then $\delta I(G_1, \iota) \subseteq \delta I(G_2, \iota)$.*

Proof.

$$\begin{aligned} w \in \delta I(G_1, \iota) &\implies pws \in L(G_1), p\iota s \in L(G_1), w \neq \iota \\ &\implies pws \in L(G_2), p\iota s \in L(G_2), w \neq \iota \\ &\implies w \in \delta I(G_2, \iota) \end{aligned}$$

So $\delta I(G_1, \iota) \subseteq \delta I(G_2, \iota)$. □

Finally, even though one may assume that intent-security for some intent length implies intent-security for lower intent length, it is not the case, as shown by the following counterexample.

Remark 6.20. *Let n, p such that $p > n \geq 1$. There exists a grammar G that is intent-secure for intent length p but not intent-secure for intent length n.*

The grammar $G = (\{a, b, c, d\}, \{S, S'\}, \{S \to S'; S \to d^n; S' \to aS'b; S' \to c^{p+1}\}, S)$ describes the language $\{d^n\} \cup \{a^k c^{p+1} b^k \mid k \geq 0\}$. This grammar is intent-secure for intent length p (the proof is similar to the one of Proposition 6.9) but is not intent-secure for intent length n as the intent d^n in the empty template (ϵ, ϵ) can be replaced by the unexpected injection c^{p+1}.

6.4. Applications to some programming languages

As expected, many of the languages commonly used are inherently intent-insecure for intent length 1. To show this, we use the Proposition 6.16 and expose templates that allow unexpected injections when the intent is a single symbol (more precisely, a token).

6.4.1. SQL

The SQL language and its multiple dialects are infamous for their injection vulnerabilities. Here is a classic example:

SELECT * FROM product WHERE price = $\sqcup$

Let us assume that the intent is a number or a string. However, unexpected injections are possible, such as `123 OR availability="true"`. This shows that the SQL language is inherently intent-insecure.

6.4.2. SMTP

SMTP (Simple Mail Transfer Protocol) is a protocol used for email transmission. It is known to be prone to injection attack in its header. Consider the following template:

```
      To: recipient@example.com
      From: ␣
      Subject:an email
       some text
      .
```

Here, the intent is a string containing an email address. However, unexpected injections are possible, such as `sender@test.com%0Acc:victim@example.com` that sends a carbon copy to someone else (the character `%0A` is used to add a new line). So the SMTP language is inherently intent-insecure.

6.4.3. LDAP

LDAP (Lightweight Directory Access Protocol) is a protocol used for directory services. It is notably used for authentication. It contain vulnerable templates, such as:

(& (USER=␣)(PASSWORD=password))

Suppose the intent is a username. Unexpected injections are possible in this template, such as `username)(&)`. So the LDAP language is inherently intent-insecure.

6.4.4. Markup languages

Markup languages are a family of languages that share a structure composed of nested tags. For example, XML documents are generally vulnerable to injection attacks, where an attacker closes and opens tags.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users> <user> <login>user1</login>
               <pwd>password1</pwd>
               <root>false<root/>
               <mail>user1@example.com</mail>
        </user>
        <user> <login>user2</login>
               <pwd>password2</pwd>
               <root>false<root/>
               <mail>␣</mail>
        </user>
</users>
```

The intent is an email address. However, unexpected injections are possible, such as:

```
 not_suspicious@example.com</mail></user>
<user><login>root_attacker</login>
<pwd>toor</pwd>
<root>true</root>
```

```
<mail>attacker@example.com
```

So the XML languages can be inherently intent-insecure. Such injections are also possible with other markup languages, such as HTML, SVG or TeX.

6.4.5. Intent-secure markup languages

The previous subsection shows an inherently intent-insecure XML language. But they are infinitely many XML languages (it is a whole language class [63]), and not all of them are inherently intent-insecure.

A simple way of constructing such an intent-secure XML language for an intent length 1 is based on the Proposition 6.9. Suppose the objective is to represent a list of pairs (key, value). A first naive, intent-insecure language relies on the following document type definition (DTD):

```
<!ELEMENT l ((k,v)*)>
<!ELEMENT k (#PCDATA)>
<!ELEMENT v (#PCDATA)>
```

This DTD can be expressed as the grammar $G = (\{<\mathbf{l}>, </\mathbf{l}>, <\mathbf{k}>, </\mathbf{k}>, <\mathbf{v}>, </\mathbf{v}>, \mathbf{s}\}, \{S, A\}, R, S)$ where $R = \{S \to <\mathbf{l}>A</\mathbf{l}>, A \to <\mathbf{k}>\mathbf{s}</\mathbf{k}><\mathbf{v}>\mathbf{s}</\mathbf{v}>A, A \to \epsilon\}$ and **s** is any text.

Considering this example:

$$<\mathbf{l}><\mathbf{k}>\textbf{first key}</\mathbf{k}>\_\_</\mathbf{l}>$$

Let us assume the intent is a tag **v**. However, unexpected injections are possible, such that <**v**>**first value**</**v**><**k**>**second key**</**k**><**v**>**second value**</**v**>. So this DTD represents a vulnerable XML language.

However, instead of chaining the pairs, we can make them nested. We get the following DTD:

```
<!ELEMENT l ((k,l,v)|(c,d))>
<!ELEMENT k (#PCDATA)>
<!ELEMENT v (#PCDATA)>
<!ELEMENT c EMPTY>
<!ELEMENT d EMPTY>
```

This DTD can be expressed as the grammar $G = (T, \{S, A\}, R, S)$ where

- $T = \{<\mathbf{l}>, </\mathbf{l}>, <\mathbf{k}>, </\mathbf{k}>, <\mathbf{v}>, </\mathbf{v}>, <\mathbf{c}/>, <\mathbf{d}/>, \mathbf{s}\}$
- $R = \{S \to <\mathbf{l}><\mathbf{k}>\mathbf{s}</\mathbf{k}>S<\mathbf{v}>\mathbf{s}</\mathbf{v}></\mathbf{l}>,$
  $S \to <\mathbf{l}><\mathbf{c}/><\mathbf{d}/></\mathbf{l}>\}$
- **s** is any text.

The previous example must be changed to comply with this second DTD.

```
<l><k>first key</k>
    <l><k>second key</k>
        <l><c/><d/></l>
        ␣
    </l>
    <v>first value</v>
</l>
```

In this example, the only possibility is to add the tag **v**: no unexpected injection is possible. The demonstration of the intent-security for intent length 1 of this grammar is similar to the proof of Proposition 6.9.

6.5. Decidability of intent-security

This subsection is dedicated to analyzing the decidability of the intent-security for classic classes of grammars: finite, regular, deterministic, and context-free. The intent-security of a grammar could greatly help the risk analysis of using that grammar and provide guarantees about its usage in critical systems.

Proposition 6.21. *The intent-security with m blanks for intent length at most n of a context-free grammar that generates a finite language is decidable with the Algorithm 1.*

Algorithm 1: Intent-security verification for finite languages

```
Input: A context-free grammar $G = (N, T, R, S)$ that generates a finite language
Input: $n \geq 1$, the maximum intent length, $m \geq 1$, the number of blanks
Output: True if $G$ is intent-secure with $m$ blanks for intent length at most $n$, False otherwise
1 Remove from $G$ all the symbols that are unreachable from $S$
2 forall $\boldsymbol{\iota} \in (\bigcup_{k=1}^{n} \Delta^k)^m$ do
3     $T_\iota \leftarrow \{\mathbf{t} \in T^* \times (T^+)^{m-1} \times T^* \mid S \Rightarrow^* \mathbf{t} \odot \boldsymbol{\iota}\}$    // Compute all possible templates
4     $I_\iota \leftarrow \{\mathbf{w} \in (T^*)^m \mid \exists \mathbf{t} \in T_\iota \ : \ S \Rightarrow^* \mathbf{t} \odot \mathbf{w}\}$    // Compute all possible injections
5     $U_\iota \leftarrow \{\mathbf{w} \in I_\iota \mid \exists i \in [1, m] : \iota_i \not\Rightarrow^* w_i\}$    // Keep the injections that are not
                                                        // explainable by the intent
6     if $U_\iota \neq \varnothing$ then return False
7 return True
```

Proof sketch. The termination of this algorithm is trivial since every operation involves finite sets. The variable U_ι is the set of injections of I_ι that cannot be derived from $\boldsymbol{\iota}$, i.e., $U_\iota = \delta I(G, \boldsymbol{\iota})$. It is empty for all $\boldsymbol{\iota}$ if and only if the grammar is intent-secure. □

Finite languages can be used in simple message protocols for embedded systems. In that case, this naive algorithm can be very useful to prove that a grammar of such language is intent-secure. However, such a result cannot be obtained for infinite regular languages. In fact, these languages are inherently insecure due to the pumping lemma.

Theorem 6.22. *Let $n \geq 1$. Infinite regular languages are inherently insecure for intent length n.*

Proof. Let L be a infinite regular language. By the pumping lemma for regular languages, there exists an integer $l > 0$ depending only on L such that every word $w \in L$ of length at least l can be written as $w = xyz$, satisfying the following conditions:

1. $|y| \geq 1$
2. $|xy| \leq l$
3. $\forall k \geq 0,\ xy^k z \in L(G)$

As $|y| \geq 1$, $|y^n| \geq n$. Let w, w' such that $|w| = n$ and $ww' = y^n$. Remark that there exist w'' such that $|w''| \geq 1$ and $ww''w' = y^{n+1}$. Let $p = x$, $s = w'z$. Then $pws = py^n z \in L$, $pww''s = py^{n+1}z \in L$ and $w \neq ww''$ so we can conclude with Proposition 6.16 that L is inherently intent-insecure for intent length n. □

This theorem may appear of little interest, as regular languages are rarely used in programming languages. However, due to the Proposition 6.19, we can conclude that any language that includes an infinite regular language is also inherently intent-insecure.

Corollary 6.23. *Let L be a formal language and L_r be an infinite grammar language such that $L_r \subseteq L$. Then L is inherently intent-insecure.*

Proof. Direct application of Proposition 6.19 and Theorem 6.22. □

In particular, if any nonterminal derives an infinite regular language in a formal grammar, then its language is inherently intent-insecure. Since such constructions are ubiquitous in programming languages (for example, any list of elements separated by commas), it explains their vulnerability to injection-based attacks. For example, the classic SQL injection `' OR '1'='1` can be viewed as an injection in the infinite regular sublanguage `SELECT * FROM table WHERE` (`<Condition> OR`)* `<Condition>` where the symbols `<Condition> OR` are pumped by the injection.

While regular infinite languages are inherently insecure, their set of unexpected injections are regular and can be easily computed. However, in most cases, the set of unexpected injections lies in a language class that is more complex than that of the query language. As an example, the following proposition shows a simple LL(1) grammar with context-sensitive unexpected injections.

Proposition 6.24. *Consider two context-free grammars $G_1 = (\{S_1, N_1\}, T, R_1, S_1)$, $G_2 = (\{S_2, N_2\}, T, R_2, S_2)$ with $T = \{k, d, a, \hat{a}, v, \hat{v}\}$ and the following rules:*

R	R_1	R_2
$S \to a\ N_2$	$S_1 \to k\ v\ N_1\ v$	$S_2 \to a\ N_2$
$S \to \hat{v}\ S_2\ v$	$N_1 \to v\ N_1\ v$	$S_2 \to \hat{v}\ S_2\ v$
$S \to v\ S_2\ \hat{v}$	$N_1 \to \hat{v}\ N_1\ \hat{v}$	$S_2 \to v\ S_2\ \hat{v}$
$S \to \hat{a}\ S_2\ a$	$N_1 \to a\ N_1\ a$	$S_2 \to \hat{a}\ S_2\ a$
$S \to S_1$	$N_1 \to \hat{a}\ N_1\ \hat{a}$	$N_2 \to a\ S_2\ \hat{a}$
	$N_1 \to d$	$N_2 \to v\ d$

Let $G = (\{S, S_1, N_1, S_2, N_2\}, T, R \cup R_1 \cup R_2, S)$. The grammar G is a LL(1) grammar and its injection language $\delta I(G, k)$ is context-sensitive and not context-free.

Proof. We first prove that G is indeed an LL(1) grammar by exhibiting its LL(1) parsing table in Table 5).

	\$	a	$\hat{v}$	v	$\hat{a}$	k	d
S		$S \to a\ N_2$	$S \to \hat{v}\ S_2\ v$	$S \to v\ S_2\ \hat{v}$	$S \to \hat{a}\ S_2\ a$	$S \to S_1$	
S_1						$S_1 \to k\ v\ N_1\ v$	
N_1		$N_1 \to a\ N_1\ a$	$N_1 \to \hat{v}\ N_1\ \hat{v}$	$N_1 \to v\ N_1\ v$	$N_1 \to \hat{a}\ N_1\ \hat{a}$		$N_1 \to d$
S_2		$S_2 \to a\ N_2$	$S_2 \to \hat{v}\ S_2\ v$	$S_2 \to v\ S_2\ \hat{v}$	$S_2 \to \hat{a}\ S_2\ a$		
N_2		$N_2 \to a\ S_2\ \hat{a}$		$N_2 \to v\ d$			

Table 5: LL(1) parsing table for the grammar G defined in Proposition 6.24

Let $L_1 = L(G_1)$ and $L_2 = L(G_2)$. The language L_1 is simply $\{kv\alpha^r d\alpha v \mid \alpha \in \{a, \hat{a}, v, \hat{v}\}^*\}$. Our goal is to compute $I(G, k)$:

$$\begin{aligned} I(G,k) &= \{z \in T^* \mid \exists p, s \in T^*, pzs \in L, S \Rightarrow^* pks\} \\ &= \{z \in T^* \mid \exists \alpha \in \{a, \hat{a}, v, \hat{v}\}^*, zv\alpha^r d\alpha v \in L\} \end{aligned}$$

This is justified by the fact that the words in L that contain k all come from L_1 (the words of L_2 cannot contain k), so necessarily $p = \epsilon$ and $s = v\alpha^r d\alpha v$.

There is only one z such that $zv\alpha^r d\alpha v \in L_1$: k. This means that the words z such that $zv\alpha^r d\alpha v \in L_2$ are exactly the unexpected injections (since $E(G, k) = \{k\}$).

$$\delta I(G,k) = \{z \in T^* \mid \exists \alpha \in \{a, \hat{a}, v, \hat{v}\}^*, zv\alpha^r d\alpha v \in L_2\}$$

In the following, the suffix palindrome ($v\alpha^r d\alpha v$) will be highlighted in blue. We want to prove that $\delta I(G, k) = \{\hat{v}a^{2^n} \mid n \geq 0\}$. To achieve this goal, we will first prove by induction on n that all words $\omega_1\omega_2$ from L_2 such that $\omega_1 v\alpha^r d\alpha v \in L_2$ must have the form: $\omega_1\omega_2 = \hat{v}a^{2^n} v(\prod_{0\leq i<n} \phi_i)^r d(\prod_{0\leq i<n} \phi_i)v$ for $n \geq 0$, where $\phi_i = va^{2^i}\hat{v}\hat{a}^{2^i}$.

In the following, γ_i will be the sequence of words of G_2 obtained by a backward (bottom-up) derivation with the rules of G_2. The previous definition of $\delta I(G, k)$ shows that our goal is to obtain a word with a suffix that is a palindrome ending with v. We start from the smallest word of $L(G_2)$, $\gamma_0 = avd$. To get a palindrome around d, we need to add v at the end of avd, and therefore use the rule $S_2 \to \hat{v}S_2v$ to obtain $\gamma_1 = \hat{v}avdv$. The sequence γ_1 ends with a palindrome vdv that ends with v, hence the injection $\hat{v}a$ as expected.

For $n = 1$, we continue to make longer words from $\gamma_1 = \hat{v}avdv$. The prefix before the palindrome vdv is $\hat{v}a$ so, to extend the palindrome, we need to add $a\hat{v}$ at the end of γ_1. It means that the sole possibility to continue is by using $S_2 \to \hat{a}S_2a$ and then $S_2 \to vS_2\hat{v}$, leading to $\gamma_2 = v\hat{a}\hat{v}avdva\hat{v}$. Because the added suffix $a\hat{v}$ does not end with v, we continue, guided by the added prefix $v\hat{a}$. We obtain $\gamma_3 = \hat{v}aav\hat{a}\hat{v}avdva\hat{v}\hat{a}v$ that ends with a v. This leads to the injection $\hat{v}aa$ as expected.

The same reasoning applies to the general case: let $\gamma_l = \hat{v}a^{2^n} v(\prod_{0\leq i<n} \phi_i)^r d(\prod_{0\leq i<n} \phi_i)v$. We are guided first by the prefix $\hat{v}a^{2^n}$, yielding: $\gamma_{l+1} = v\hat{a}^{2^n}\gamma_l a^{2^n}\hat{v}$. Since γ_{l+1} does not end with v, we are then guided by the starting sequence $v\hat{a}^{2^n}$. We get: $\gamma_{l+2} = \hat{v}(aa)^{2^n}\gamma_{l+1}\hat{a}^{2^n}v$. Written completely, it is:

$$\gamma_{l+2} = \hat{v}(aa)^{2^n} v\hat{a}^{2^n}\hat{v}a^{2^n} v(\prod_{0\leq i<n} \phi_i)^r d(\prod_{0\leq i<n} \phi_i)va^{2^n}\hat{v}\hat{a}^{2^n}v$$

As $\phi_n = va^{2^n}\hat{v}\hat{a}^{2^n}$, γ_{l+2} can finally be rewritten into:

$$\gamma_{l+2} = \hat{v}a^{2^{n+1}} v(\prod_{0\leq i<n+1} \phi_i)^r d(\prod_{0\leq i<n+1} \phi_i)v$$

This completes the induction. In the end, $\delta I(G,k) = \{\hat{v}a^{2^i} \mid i \geq 0\}$: this language is well known to be context-sensitive but not context-free [50]. □

So the languages of unexpected injection can be much more complex than the initial grammar: in this case, an LL(1) grammar can yield a context-sensitive set of unexpected injections. Since the emptiness is undecidable for context-sensitive languages, this example suggests that the intent-security of a grammar (verified by the emptiness of its unexpected injection set) could be undecidable even for simple grammars. In fact, the next theorems show that the unexpected injection languages of LR(0) grammars can be any recursively enumerable languages, and so that their intent-security is not decidable.

Our theorems are based on the following lemma.

Lemma 6.25. *For every recursively enumerable language L, there exist a regular set $\mathcal{R}$ and two morphisms g and h, such that*

$$L = \{g(w) \diagup h(w) \mid w \in \mathcal{R}\}$$

Proof. This proof is based on the proof of a similar theorem in [64]. We assume that L is defined by a grammar $G = (N, T, R, S)$ and that each rule in R is identified by a unique symbol: $R = \{r_i : \alpha_i \to \beta_i\}_{i \in I}$, where I is a finite index set. In addition to symbols from T and N, the authors introduce $|I| + 3$ new symbols, namely A, B, $\overline{B}$, and the symbols $\{r_i\}_{i \in I}$. The regular set $\mathcal{R}^1_G = A(B\Delta^* R \Delta^*)^* \overline{B}$ is defined, as well as the homomorphisms described in Table 6.

	A	B	$\overline{B}$	a	r_i
g_G	ABS	B	ϵ	a	β_i
h_G	A	B	B	a	α_i

Table 6: Morphisms g_G and h_G with $a \in \Delta$ and $r_i : \alpha_i \to \beta_i \in R$

We focus on the intuition behind $\mathcal{R}^1_G$. Consider the following derivation:

$$\begin{aligned} S = \alpha_a &\Rightarrow^{r_a} \beta_a = u_1 \alpha_b v_1 \\ &\Rightarrow^{r_b} u_1 \beta_b v_1 = u_2 \alpha_c v_2 \\ &\Rightarrow^{r_c} u_2 \beta_c v_2 \end{aligned}$$

where $\Rightarrow^{r}$ indicates that the rule r has been applied, and u_i, v_i indicate in which context the rule has been applied. Let consider the word $w = ABr_aBu_1r_bv_1Bu_2r_cv_2\overline{B}$ that belongs to $\mathcal{R}^1_L$ and its image though g_G and h_G:

$$\begin{aligned} w &= A \quad\;\;\; Br_a \; Bu_1r_bv_1 \; Bu_2r_cv_2 \; \overline{B} \\ g_G(w) &= ABS \; B\beta_a \; Bu_1\beta_bv_1 \; Bu_2\beta_cv_2 \\ h_G(w) &= A \quad\;\;\; B\alpha_a \; Bu_1\alpha_bv_1 \; Bu_2\alpha_cv_2 \; B \end{aligned}$$

Remark that $h_G(w)$ is a prefix of $g_G(w)$: in this example, $S = \alpha_a$, $\beta_a = u_1\alpha_bv_1$ and $u_1\beta_bv_1 = u_2\alpha_cv_2$. In the end, $h_G(w) \diagdown g_G(w)$ is the last sentential form of the derivation, in this case $u_2\beta_cv_2$. The quotient ensure that the derivation is valid. The theorem of [64] states that this quotient, when intersected with T^*, recreates exactly the words of the initial language and only those, i.e.,

$$L = \{h_G(w) \diagdown g_G(w) \mid w \in \mathcal{R}^1_G\} \cap T^*$$

However, we can slightly modify this proof to omit this intersection with T^*. In fact, this intersection is used to rule out the intermediate forms that are not words. But, by modifying $\mathcal{R}^1_G$, we can ensure that the quotient result is always a word. First, let show that the previous theorem holds if we replace $\mathcal{R}^1_G = A(B\Delta^* R\Delta^*)^*\overline{B}$ by $\mathcal{R}^2_G = A(B\Delta^* R\Delta^*)^+\overline{B}$:

$$\begin{aligned}
L &= \{h_G(w) \diagdown g_G(w) \mid w \in \mathcal{R}^1_G\} \cap T^* \\
&= (\{h_G(w) \diagdown g_G(w) \mid w \in A(B\Delta^* R\Delta^*)^+\overline{B}\} \cap T^*) \\
&\qquad \cup (\{h_G(A\overline{B}) \diagdown g_G(A\overline{B})\} \cap T^*) \\
&= (\{h_G(w) \diagdown g_G(w) \mid w \in A(B\Delta^* R\Delta^*)^+\overline{B}\} \cap T^*) \\
&\qquad \cup (\{S\} \cap T^*) \\
&= \{h_G(w) \diagdown g_G(w) \mid w \in \mathcal{R}^2_G\} \cap T^*
\end{aligned}$$

To ensure that all derivations will end with words, let's define the final rules R_f as the rules of R whose right-hand side is a word: $R_f = \{r_i : \alpha_i \to \beta_i \in R \mid \beta_i \in T^*\}$. Then, we can construct our new regular set $\mathcal{R}^3_G = A(B\Delta^* R\Delta^*)^*(BT^* R_f T^*)\overline{B}$. Let show that the intersection with T^* is useless with this regular set by showing that $\forall w \in \mathcal{R}^3_G, h_G(w) \diagdown g_G(w) \in T^*$.

Let $w \in \mathcal{R}^3_G$ and let $u, v \in T^*$ and $r : \alpha \to \beta \in R_f$ (so $\beta \in T^*$) such that $w \in A(B\Delta^* R\Delta^*)^*(Burv)\overline{B}$. As $h_G(w) \diagdown g_G(w) = u\beta v \in T^*$, we can conclude that $\{h_G(w) \diagdown g_G(w) \mid w \in \mathcal{R}^3_G\} \subseteq T^*$.

Remark that $\mathcal{R}^3_G \subseteq \mathcal{R}^2_G$ since $T \subset \Delta$ and $R_f \subseteq R$. Therefore:

$$\begin{aligned}
&\{h_G(w) \diagdown g_G(w) \mid w \in \mathcal{R}^3_G\} \subseteq \\
&\qquad\qquad \{h_G(w) \diagdown g_G(w) \mid w \in \mathcal{R}^2_G\} \cap T^* = L
\end{aligned}$$

What is left to prove is that $L \subseteq \{h_G(w) \diagdown g_G(w) \mid w \in \mathcal{R}^3_G\}$. Let w_G be a word of L. There exists $w \in \mathcal{R}^2_G$ such that $w_G = h_G(w) \diagdown g_G(w)$. Let show that $w \in \mathcal{R}^3_G$. Let $u, v \in \Delta^*$, $r : \alpha \to \beta \in R$ such that $w = A(B\Delta^* R\Delta^*)^*(Burv)\overline{B}$. We know that $w_G = h_G(w) \diagdown g_G(w) = u\beta v$. As w_G is in T^*, $u\beta v$ is in T^*, so $u, v \in T^*$ and the right-side production of r is in T^*, i.e., $r \in R_f$. Finally, we showed that $w \in \mathcal{R}^3_G$ and thus $w_G \in \{h_G(w) \diagdown g_G(w) \mid w \in \mathcal{R}^3_G\}$.

We can therefore conclude that $L = \{h_G(w) \diagdown g_G(w) \mid w \in \mathcal{R}^3_G\}$.

Let $G' = (N, T, R', S)$ such that $\alpha \to \beta \in R'$ iff $\alpha^r \to \beta^r \in R$, so $L(G') = L^r$. We use G' to end up with the equation presented in the theorem:

$$\begin{aligned}
L &= (L^r)^r \\
&= \{h_{G'}(w) \diagdown g_{G'}(w) \mid w \in \mathcal{R}^3_{G'}\}^r \\
&= \{m \mid h_{G'}(w)m = g_{G'}(w), w \in \mathcal{R}^3_{G'}\}^r \\
&= \{m^r \mid h_{G'}(w)m = g_{G'}(w), w \in \mathcal{R}^3_{G'}\} \\
&= \{m^r \mid m^r h_{G'}(w)^r = g_{G'}(w)^r, w \in \mathcal{R}^3_{G'}\} \\
&= \{g_{G'}(w)^r \diagup h_{G'}(w)^r \mid w \in \mathcal{R}^3_{G'}\} \\
&= \{g^r_{G'}(w) \diagup h^r_{G'}(w) \mid w \in (\mathcal{R}^3_{G'})^r\}
\end{aligned}$$

where $g^r_{G'}$ and $h^r_{G'}$ are defined in Table 7.

□

We can now prove the main result.

	A	B	$\overline{B}$	a	r_i
$g^r_{G'}$	SBA	B	ϵ	a	β^r_i
$h^r_{G'}$	A	B	B	a	α^r_i

Table 7: Morphisms $g^r_{G'}$ and $h^r_{G'}$ with $a \in \Delta$ and $r_i : \alpha_i \to \beta_i \in R'$

Theorem 6.26. *Let $n \geq 1$. A language L is recursively enumerable if and only if there exists an LR(0) grammar G such that $L \in \delta\mathcal{I}^n(G)$.*

Proof. This result is based the previous Lemma. Let us show first the result for $n = 1$. Let $\mathcal{R} \subseteq T^*$, $g : T \to T$ and $h : T \to T$ such that $L = \{g(w) \diagup h(w) \mid w \in \mathcal{R}\}$ as defined in the proof of Lemma 6.25. Our goal is to create a deterministic context-free grammar G and a symbol # such that its unexpected injection language $\delta\mathcal{I}(G, \#)$ is in the form $g(w) \diagup h(w)$. To this end, we introduce three new languages, L_1, L_2 and L_3, defined over the symbols $T' = T \cup \{\#, \$\}$ as:

$$L_1 = \{w^r \ \$ \ \# \ h(w) \mid w \in \mathcal{R}\}$$
$$L_2 = \{w^r \ \$ \ g(w) \mid w \in \mathcal{R}\}$$
$$L_3 = L_1 \cup L_2$$

Let's show that L_3 is a deterministic context-free language by constructing a deterministic pushdown automaton (DPDA) that recognizes it. Let A_r be a deterministic finite-state automaton that recognizes the regular language $\mathcal{R}^r$ and let modify it into A'_r such that each symbol that is read is pushed to the stack. Let A_g (resp. A_h) be a DPDA that recognizes $g(w)$ (resp. $h(w)$) by empty stack guided by the word w^r that has already been pushed. More precisely, denote q_g the initial state of A_g and A'_g the rest of the states of A_g.

During the processing of the word, we can detect the end of the regular expression when the symbol \$ (absent in $\mathcal{R}$) occurs. The only tricky part is to choose which DPDA to use after \$, either A_h or continuing with A'_g. It can be easily done thanks to the symbol # since no word of $g(w)$ contains # (cf. the form of $\mathcal{R}$ and Table 6). Hence, if # is encountered, we continue with A_h and otherwise, we continue with A_g. A DPDA for L_3 can be built as shown in Figure 2: the words of L_3 are accepted by empty stack.

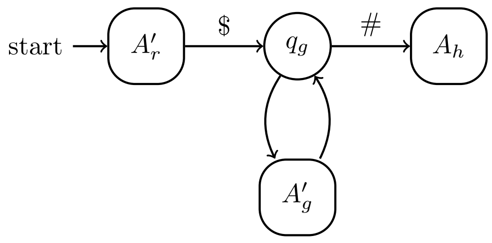

$$\begin{tikzcd}
\text{start} \arrow[r] & A'_r \arrow[r, "\$"] & q_g \arrow[r, "\#"] \arrow[d, bend right] & A_h \\
 & & A'_g \arrow[u, bend right] &
\end{tikzcd}$$

Figure 2: A summarized DPDA for the language L_3

Now that we know that L_3 is a deterministic context-free language, we examine its unexpected injection language at #.

$$\begin{aligned}
\delta I(G,\#) &= \{m \in T'^* \mid \exists p, s \in T'^*, pms \in L_3, p\#s \in L(G)\} - E(G,\#) && (1)\\
&= \{m \in T'^* \mid \exists w \in \mathcal{R}, w^r\$mh(w) \in L_3\} - \{\#\} && (2)\\
&= \{w^r\$\diagdown m \diagup h(w) \mid w \in \mathcal{R}, m \in L_3\} - \{\#\} && (3)\\
&= (\{w^r\$\diagdown m \diagup h(w) \mid w \in \mathcal{R}, m \in L_2\} \cup \{\#\}) - \{\#\} && (4)\\
&= \{w^r\$\diagdown w_2^r\$g(w_2) \diagup h(w) \mid w, w_2 \in \mathcal{R}\} && (5)\\
&= \{g(w) \diagup h(w) \mid w \in \mathcal{R}\} && (6)\\
&= L && (7)
\end{aligned}$$

Passing from line (1) to line (2) is justified by the fact that all the words in L_3 that contain # come from L_1 (since the words of L_2 cannot contain #), so necessarily $p = w^r\$$ and $s = h(w)$ for any $w \in \mathcal{R}$. Passing from line (3) to line (4) is justified by the fact that $\{w^r\$\diagdown m \diagup h(w) \mid w \in \mathcal{R}, m \in L_1\} = \{\#\}$. Passing from line (6) to line (7) is justified by the Lemma 6.25.

Finally, we constructed the deterministic context-free language L_3 with a deterministic grammar G such that $\delta I(G,\#) = L$. So $L \in \delta\mathcal{I}^1(G)$.

In fact, we can expand this theorem: first, the grammar G we construct is in fact LR(0) and second, we can modify the proof to replace $\delta\mathcal{I}^1(G)$ by any $\delta\mathcal{I}^n(G)$ for a fixed value of $n \geq 1$.

This is a direct consequence of a result shown by [53]: LR(0) languages are exactly the languages recognized by a deterministic pushdown automaton that accepts by empty stack. This is the case of the automaton constructed in the previous proof, so L_3 is an LR(0) language. In fact, we can modify the construction of L_1 in the previous proof by replacing # with $\#^n$ (n times the symbol #) and therefore get that $\delta I(G,\#^n) = L$. So, for any $n \geq 1$, a language L is recursively enumerable if and only if there exists an LR(0) grammar G and $\iota \in \Delta^n$ such that $L = \delta I(G,\iota)$. □

The undecidability of the intent-security is a corollary of the previous theorem and the fact that emptiness is undecidable for recursively enumerable languages.

Corollary 6.27. *Let G be an LR(0) grammar, $n \geq 1$. It is undecidable whether G is intent-secure for intent length n.*

Proof. Since the emptiness of recursively enumerable languages is undecidable, it is undecidable whether $\delta I(G,\iota) = \varnothing$ for any $\iota \in \Delta^+$. Therefore, it is undecidable whether G is intent-secure for intent length n. □

If the intent-security is undecidable for the class of LR(0) grammars, it is also undecidable for the classes that include LR(0) grammars, such as LR(k) grammars, deterministic grammars and context-free grammars. We show a similar result for another subclass of context-free grammars: linear grammars.

Theorem 6.28. *Let $n \geq 1$. A language L is recursively enumerable if and only if there exists a linear grammar G such that $L \in \delta\mathcal{I}^n(G)$.*

Proof. Let us show that we can construct a linear grammar G for some recursively enumerable language L. As shown by Latteux and Turakainen [64] (Corollary 4), there exist two linear grammars $G_1 = (N_1, T, R_1, S_1)$ and $G_2 = (N_2, T, R_2, S_2)$ such as $L = L(G_2)\diagdown L(G_1)$. Let us rewrite this quotient as the set of unexpected injections of a grammar G.

To this end, we will first modify G_1 to introduce a marker \$. First, let us introduce a fresh terminal \$, and let $h : T \cup \{\$\} \to T$ be the homomorphism such that $h(\$) = \epsilon$ and otherwise $h(t) = t$ for all $t \in T$. Consider the language $L' = h^{-1}(L(G_1)) \cap (T^*\$T^*) = \{w\$v \mid wv \in L(G_1)\}$. Because of the closure of linear languages for inverse homomorphism and intersection with a regular language, $L(G_1)$ is linear. Let G_1' be a linear grammar for $L(G_1)$. Let $\#$ be a new symbol, and consider the linear grammar $G = (N_1 \cup N_2, T \cup \{\$, \#\}, R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\$\#\}, S)$. Note that $L(G) = L(G_1') \cup (L(G_2) \cdot \$\#)$. We show that $\delta I(G, \#) = L(G_2) \diagdown L(G_1)$. By definition of G, the only templates where $\#$ can appear are $\{(v\$, \epsilon) \mid v \in L(G_2)\}$. Then:

$$\begin{aligned} I(G,\#) &= \{w \in T^* \mid v\$w \in L(G), v \in L(G_2)\} \\ &= \{w \in T^* \mid v\$w \in L(G_1'), v \in L(G_2)\} \cup \{\#\} \\ &= \{w \in T^* \mid vw \in L(G_1), v \in L(G_2)\} \cup \{\#\} \\ &= (L(G_2) \diagdown L(G_1)) \cup \{\#\} \end{aligned}$$

Since $\#$ is a terminal symbol, we have $E(G, \#) = \{\#\}$. Then:

$$\delta I(G,\#) = I(G,\#) - E(G,\#) = L(G_2) \diagdown L(G_1) = L$$

In fact, we can modify the construction of G by replacing $\#$ with $\#^n$ and therefore get that $\delta I(G, \#^n) = L$. So, L is recursively enumerable if and only if there exists a linear grammar G such that $L \in \delta\mathcal{I}^n(G)$. □

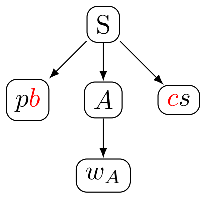
Corollary 6.29. *Let G be a linear grammar, $n \geq 1$. It is undecidable whether G is intent-secure for intent length n.*

Proof. Same as Corollary 6.27. □

These results show there is no automatic way of proving that some LR(0) or linear grammar is intent-secure. Since LR(0) are among the most simple deterministic grammars, it means that the vast majority of programming languages would be difficult to be proved intent-secure. For such languages, the static analysis of individual templates (discussed in section 5) is a more promising approach.

The last result was shown for one blank. In the following, we show that infinite context-free languages are inherently intent-insecure for at least two blanks.

Theorem 6.30. *Infinite context-free languages are inherently intent-insecure for at least two blanks.*


(a) Simplified version of the parse tree of pbw_Acs

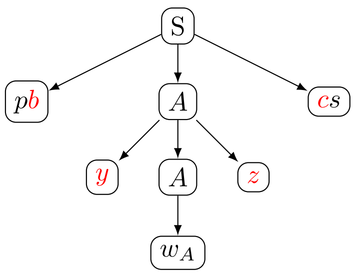

(b) Simplified version of the parse tree of $pbyw_Azcs$

Figure 3: Illustration of the proof of Theorem 6.30. Injections are in red.

Proof. Let G be a context-free grammar that describes an infinite language. If G describes a regular language, then according to Theorem 6.22 it is not intent-secure with a single blank, so it cannot be intent-secure with two blanks.

Let us assume that G describes a non-regular language. As remarked by Chomsky in [48], G must be self-embedding, which means there is a symbol $A \in N$ and $y, z \in T^+$ (so $y \neq \epsilon$ and $z \neq \epsilon$) such that $A \Rightarrow^* yAz$. Let denote b the last terminal of y and c the first terminal of z. Let $\iota_1 = b$ and $\iota_2 = c$. Let $w_1, w_2 \in T^*$ such that $S \Rightarrow^* w_1 A w_2$. Let's define $p, s \in T^*$ such that $pb = w_1 y$ and $cs = z w_2$. Then $S \Rightarrow^* w_1 A w_2 \Rightarrow^* w_1 y A z w_2 = pbAcs$.

Let $w_A \in T^+$ such that $A \Rightarrow^* w_A$. Let consider the following template $p\_\__1 w_A \_\__2 s$ where the first (resp. second) blank is associated with the intent b (resp. c). This template, filled with the intents, leads to pbw_Acs, a word in $L(G)$. Let consider the injection (by, zc). Because $S \Rightarrow^* pbAcs$ and $pbAcs \Rightarrow^* pbyAzcs \Rightarrow^* pbyw_Azcs$, the latter is in $L(G)$. So $(by, zc) \in I(G, (b, c))$. This is illustrated by Figure 3.

Since $y \neq \epsilon$ and $z \neq \epsilon$, $by \neq b$ and $zc \neq c$ so $(by, zc) \notin \mathcal{E}((b, c)) = \{(b, c)\}$. Finally, we can conclude that $(by, zc) \in \delta I(G, (b, c))$. and therefore that G is inherently intent-insecure with two blanks. Obviously, the same example can be carried out if there are more than two blanks. □

Such recursive structure is ubiquitous in many languages. This theorem shows that if there is a blank on both side of such recursive structure in a template, then it is not safe. As an example, the simple intent-secure language for one blank proposed in Proposition 6.9 ($\{a^n cdb^n \mid n \geq 0\}$) is vulnerable with two blanks: in the template $a\_\__1 acdb\_\__2 b$, where the intent is a for $\_\__1$ and b for $\_\__2$, the injection (aa, bb) is valid as well.

Remark 6.31. *Our definitions are not adapted to non-terminal intents for context-sensitive grammars. However, we would like to point out an interesting result: there probably exist context-sensitive languages that are not inherently intent-insecure for several blanks. Consider a context-free languages L and the language L'_k defined as (where # is a new symbol not present in L):*

$$L'_k = \{w(\#\#w)^k \mid w \in L\}$$

L'_k is a variant of the copy language known to be context-sensitive but not context-free [50]. We claim that L'_k is intent-secure for up to k blanks for intents composed of one terminal (i.e., $\iota_i \in T$ for $1 \leq i \leq k$). Let us give the intuition of this result for $k = 2$. In this case, $L'_2 = \{w\#\#w\#\#w \mid w \in L\}$. On one hand, the word w is repeated three times. On the other hand, the attacker can modify at most two of these words because of the two blanks (for a reason similar to what happens in Prop. 6.9). Modifying two of the three w leads to a grammatical error because the three words w won't be the same. The same idea can be expanded to any number of blanks, as long as there is at least one extra copy of w.

Remark that L'_k is built around the context-free language L: such a construction could be used to protect an existing language, even though this construction is very long.

We can conclude that the only context-free grammars that are intent-secure with at least two blanks are finite languages. This fact is important to consider when designing a new network protocol. For example: dealing with size-bounded messages (or adding context-sensitive fields such as payload length) is a necessary measure for injection-vulnerability-free protocols. We expect context-sensitive languages to allow for stronger intent-security guarantees, but this question is out of the scope of our study.

7. Characterization of unexpected injections

We presented in the previous sections two security properties: the intent-equivalence of a template and the intent-security of a grammar. Another type of result we are interested in is the characterization of the set of unexpected injections for whole classes of grammars. So we define the set of unexpected injections on a set of grammars as the union of all grammars and intent length.

Definition 7.1 (Unexpected injections on a set of grammars)**.** Let $\mathcal{C}$ be a set of grammars. We define the set of languages of <u>unexpected injections on $\mathcal{C}$</u> as:

$$\delta\mathcal{I}(\mathcal{C}) = \bigcup_{G\in\mathcal{C}} \bigcup_{n>0} \delta\mathcal{I}^n(G)$$

First, we show that we can characterize the set of unexpected injections of regular languages: a grammar is regular if and only if it can be expressed as the set of unexpected injections of a regular grammar.

Theorem 7.2. *$REGL = \delta\mathcal{I}(REGG)$*

Proof. Let us first show that REGL $\subseteq \delta\mathcal{I}$(REGG). Let L be a regular language and $G = (N, T, R, S)$ a regular grammar of L. Let $G' = (N \cup \{S'\}, T \cup \{\#\}, R \cup \{S' \to S, S' \to \#\}, S')$. G' is a regular grammar. Remark that $L(G') = L \cup \{\#\}$. Then $I(G', \#) = L \cup \{\#\}$ and $E(G', \#) = \{\#\}$, so $L = I(G', \#) - E(G', \#) = \delta I(G', \#)$. So REGL $\subseteq \delta\mathcal{I}$(REGG).

Let $G = (N, T, R, S)$ be a regular grammar and $\iota \in \Delta^n$. Let us show that $\delta I(G, \iota)$ is regular. Let us consider two cases: ι contains at least one nonterminal and $\iota \in T^*$. Recall that $\delta I(G, \iota) = I(G, \iota) - E(G, \iota)$. The language $E(G, \iota)$ is regular and set difference is closed for regular language. Thus, to prove that $\delta I(G, \iota)$ is regular, we only need to prove that $I(G, \iota)$ is regular.

In the first case, ι contains at least one nonterminal. Due to the form of the rules in the regular grammar, every sentential form derivable from S is either in T^* or in $T^* \times N$. Thus, $\iota \in T^* \times N$ and all templates where ι can appear have an empty suffix. Let us construct the regular grammar G' that describes the set of sentential form of G: $G' = (T \cup \{A' \mid A \in N\}, N, R \cup \{A \to A' \mid A \in N\}, S)$. So $G' = \{\alpha \mid \alpha \in \Delta^* : S \Rightarrow^*_G \alpha\}$. Due to the form of its rules, G' is regular.

By definition, $I(G, \iota) = \{w \in T^* \mid \exists p, s \in T^* : pws \in L(G) \text{ and } S \Rightarrow^* p\iota s\}$. However, we know that the suffix is empty so $I(G, \iota) = \{w \in T^* \mid \exists p \in T^* : pw \in L(G) \text{ and } S \Rightarrow^* p\iota\}$. Let $P = L(G')\diagup\iota$ the prefixes of sentential forms of G that end with ι. Thus, $I(G, \iota) = \{w \in T^* \mid \exists p \in P : pw \in L(G)\} = P\diagdown L(G) = (L(G')\diagup\iota)\diagdown L(G)$. This quotient leads to a regular language. Finally, $I(G, \iota)$ is regular.

In the second case, $\iota \in T^*$. For every (p, s) such as $S \Rightarrow^* p\iota s$, $F(L, (p, s))$ is regular. We will show that $I(G, \iota) = \bigcup_{\{(p,s) \mid S \Rightarrow^* p\iota s\}} F(L, (p, s))$ is in fact a finite union and therefore that $I(G, \iota)$ is regular. Denote A a deterministic finite state machine with k states that recognizes $L(G)$. After reading the prefix p, A can be in one of k states. Denote this state $S_{start}(p)$. For any s, there exist none, one or multiple states from which reading s lead to a final state. Denote this set of states $S_{end}(s)$.

Let us prove that for any $(p, s), (p', s')$ such as $S_{start}(p) = S_{start}(p')$ and $S_{end}(s) = S_{end}(s')$, we have $F(L, (p, s)) = F(L, (p', s'))$. Let $A_{(p,s)}$ be the same automata as A with the difference that its initial state $S_{start}(p)$ and its finale states are $S_{end}(s)$. We define in a similar fashion $A_{(p',s')}$. Then $w \in F(L, (p, s))$ if and only if pws is recognizable from A. In that case, after reading p, A is in state $S_{start}(p)$ and after reading w, it is in one of the state of $S_{end}(s)$. So $A_{(p,s)}$ recognizes w. If $S_{start}(p) = S_{start}(p')$ and $S_{end}(s) = S_{end}(s')$, then $A_{(p',s')} = A_{(p,s)}$ and so $A_{(p',s')}$ recognizes w as well. Therefore $w \in F(L, (p', s'))$.

So $F(L,(p,s)) = F(L,(p',s'))$. It means that $I(G,\iota)$ is a finite union (of at most $k2^k$ sets) of regular languages, so $I(G,\iota)$ is regular.

Finally, in both cases, $\delta I(G,\iota) = I(G,\iota) - E(G,\iota)$ is regular so $\delta\mathcal{I}(\text{REGG}) \subseteq \text{REGL}$.

Finally, $\text{REGL} = \delta\mathcal{I}(\text{REGG})$. □

Based on the results from the last subsection, we can also show that any recursively enumerable language can be expressed as the set of unexpected injection of an LR(0) grammar.

Theorem 7.3. *$RE = \delta\mathcal{I}(LR(0))$*

Proof. Let us first prove that $\delta\mathcal{I}(\text{CFG}) \subseteq \text{RE}$. For any $G \in \text{CFG}, n \geq 0$ and $\iota \in \Delta^n$, the following equality is verified: $\delta I(G,\iota) = I(G,\iota) - E(G,\iota)$.

Let us first show that $I(G,\iota)$ is recursively enumerable. Let $w \in T^*$. There exists an algorithm that halts when $w \in I(G,\iota)$: one can iterate over all templates $(p,s) \in (T^*)^2$ and verify whether $S \Rightarrow^* p\iota s$ and $S \Rightarrow^* pws$ (this is decidable because G is context-free). Therefore $I(G,\iota)$ is recursively enumerable.

Because G is a context-free grammar, $E(G,\iota)$ is also a context-free language recognized by the grammar $G' = (N \cup \{S'\}, T, R \cup \{S' \to \iota\}, S')$, with S' a new nonterminal. It means in particular that $E(G,\iota)$ is a decidable language and therefore that $\overline{E(G,\iota)}$ is also decidable, hence recursively enumerable.

Finally, $\delta I(G,\iota) = I(G,\iota) - E(G,\iota) = I(G,\iota) \cap \overline{E(G,\iota)}$ is the intersection of two recursively enumerable languages and is therefore itself a recursively enumerable language. So $\delta\mathcal{I}(\text{CFG}) \subseteq \text{RE}$. Besides, $\delta\mathcal{I}(\text{LR(0)}) \subseteq \delta\mathcal{I}(\text{CFG})$ (Theorem 6.17) so $\text{RE} \supseteq \delta\mathcal{I}(\text{LR(0)})$.

Let prove that $\text{RE} \subseteq \delta\mathcal{I}(\text{LR(0)})$. Let $L \in \text{RE}$ and $n \geq 0$. Due to Theorem 6.26, there exists an LR(0) grammar G such as $L \in \delta\mathcal{I}^1(G)$. So $L \in \delta\mathcal{I}(\text{LR(0)})$. Therefore, $\text{RE} \subseteq \delta\mathcal{I}(\text{LR(0)})$.

Finally, $\text{RE} = \delta\mathcal{I}(\text{LR(0)})$. □

Theorem 7.4. *$RE = \delta\mathcal{I}(LinG)$*

Proof. The proof is similar to the previous one but relies on Theorem 6.28 instead. □

This concludes our analysis of intent-equivalence, intent-security and unexpected injections characterization.

8. Conclusion

Injections are among the most common threats to online services. Even if countermeasures exist, there is no guarantee that a developer will always have the knowledge or time to implement them sufficiently, if they are aware of the problem at all. Furthermore, to the best of our knowledge, no research has been done on protections embedded in the language itself.

In this paper, we formalized the notions of intent-secure grammar and intent-equivalent query. If a grammar is intent-secure, the developer is sure that only expected injections (according to their intent) can lead to grammatically correct sentences, as long as their intent consists of a single symbol (terminal or not). This property is very strong since the developers do not have to specify their intent: they do not have to be an expert in the grammar of the languages they use. As a "sanity check", we applied our definition to well-known language to show that they are indeed inherently intent-insecure. Besides, we showed as well that infinite intent-secure languages do exist.

$$\begin{tikzcd}
\text{REGL} = \delta\mathcal{I}(\text{REGG}) \arrow[r, "\subsetneq"] & \text{CFL} \arrow[r, "\subsetneq"] & \text{RE} = \delta\mathcal{I}(\text{LR}(0)) = \delta\mathcal{I}(\text{LinG})\ \delta\mathcal{I}(\text{CFG}) \\
& \delta\mathcal{I}(\text{LL}(1)) \arrow[u, "\not\subseteq"] \arrow[ur, "\subseteq"'] &
\end{tikzcd}$$

Figure 4: Language inclusion.

However, such property is not easy to obtain. The results on unexpected injection languages are closely related to the $\delta\mathcal{I}(\mathcal{C})$ sets, whose relationships are summarized in Figure 4. Grammars of finite languages are easy to prove to be intent-secure. Infinite regular languages (and their supersets) are inherently insecure. The languages of unexpected injections of LR(0) and linear grammars are exactly the recursively enumerable languages and thus the intent-secure property of a deterministic grammar is undecidable. Furthermore, every infinite context-free language is vulnerable with multiple blanks. The intent-equivalence for a given query is also undecidable for context-free grammars but, fortunately, it is decidable for LR(0) grammars and visibly-pushdown grammars (for all intent) and LR(k) grammars (for intents of length 1). Decidability results on intent-equivalence are summarized in Table 8 and results on intent-security are summarized in Table 9.

	One or more blanks $\boldsymbol{\iota} \in (\Delta)^m$	One or more blanks $\boldsymbol{\iota} \in (\Delta^+)^m$	One or more blanks $\boldsymbol{\iota} \in (T^+)^m$
REGG	D (Th. 5.12)	D (Th. 5.12)	D (Th. 5.19)
VPG	D (Th. 5.16)	D (Th. 5.16)	D (Th. 5.19)
LR(0)	D (Th. 5.15)	D (Th. 5.15)	D (Th. 5.19)
LR(k)	D (Th. 5.14)	?	D (Th. 5.19)
LinG	U (Th. 5.17)	U (Th. 5.17)	D (Th. 5.19)
CFG	U (Co. 5.18)	U (Co. 5.18)	D (Th. 5.19)

Table 8: Intent-equivalence property. D: decidable. U: undecidable. ?: unknown

Thus, these results open up many theoretical possibilities and technical implications. In the following, we identify some of them, including the ones presented in the introduction.

Q1: *Does string concatenation necessarily lead to injection vulnerabilities?* String concatenation is generally recognized as a major source of injection vulnerabilities [2]. Popular protections like prepared statements aim to prevent the developer from directly performing such concatenation. Our study is another proof that string concatenation can indeed easily bring injection vulnerabilities. However, we also showed

	One blank	Two or more blanks
Finite, $|L| \geq 2$	D (Pr. 6.21)	D (Pr. 6.21)
REGG with infinite language	F (Th. 6.22)	F (Th. 6.22)
Context-free grammars with infinite regular sublanguage	F (Co. 6.23)	F (Th. 6.30)
LR(0) with infinite language	U (Co. 6.27)	F (Th. 6.30)
LinG with infinite language	U (Co. 6.29)	F (Th. 6.30)
CFG with infinite language	U (Co. 6.27)	F (Th. 6.30)

Table 9: Intent-security property. F: always false. D: decidable. U: undecidable.

that string concatenation with only one user input does not *necessarily* lead to injection vulnerabilities as even complex grammars can be intent-secure (Corollary 6.27).

Q2: *Could a skilled developer write only secure queries?* Injection vulnerabilities are often considered to stem from bad programming practices. This reasoning implicitly assumes that a skilled developer could always write secure queries, or at least verify that their queries are not vulnerable to injection-based attacks. Such a position is notably supported in the testing community [65]: "data validation is the first line of defense against a hostile world". However, Theorems 6.26 and 7.3 and the Corollary 6.27 show that malicious injections may be very hard to detect. Moreover, languages with intricate unexpected injections are not necessarily exotic or unsuitable for computer science, as shown by the Proposition 6.24. This proposition considers an LL(1) language with an XML-like structure whose language of unexpected injections for some intent is context-sensitive. It shows that the injection vulnerabilities are embedded in the language itself and that one cannot rely on just the developer's skills to avoid them. If one uses an intent-insecure grammar, additional security mechanisms, such as filtering, input sanitization, static analysis, or intrusion detection, are essential.

Q3: *Is a simpler language always more secure than a more complex one?* It is generally believed that simpler languages are less prone to being attacked, as explained in [66]: "a complex computational system is an engine for executing malicious computer programs delivered in the form of crafted input". We confirmed the classic observation that a language subset is at least as secure as the whole language (Theorem 6.17). Simple languages are very useful as the intent-security and intent-equivalence of finite languages is decidable (Proposition 6.21). However, we would like to add some nuance to that statement: we also showed that a complex language is not necessarily less safe than a simpler language. For example, all infinite regular languages are intent-insecure (Theorem 6.22), while context-free languages may be intent-secure (Proposition 6.9). Furthermore, context-free languages are inherently intent-insecure with two blanks but we expect some context-sensitive grammars to be intent-secure with multiple blanks, as expressed in Remark 6.31.

Q4: *Why do (nearly) all computer languages contains injection vulnerabilities?* As shown by Corollary 6.23, every language that is a superset of an infinite regular language is inherently intent-insecure for a simple intent. This result has great significance, as programming languages almost always include such regular patterns: SQL includes the infinite regular language `SELECT` (`column,`)* `column FROM table WHERE condition` while programming and scripting languages generally accept statement lists in the form (`statement;`)* `statement`. Arithmetic and Boolean expression generally include infinite regular languages as well, e.g. (`3 +`)* `5` and (`b OR`)* `b`. This kind of regular structure is widely used because of its simplicity but allows for injection attacks. It would undoubtedly be challenging to replace these structures with non-regular patterns while still retaining their simplicity for the developers. But even in the absence of such patterns, it is impossible to avoid injection vulnerabilities in some queries with two blanks in usual computer languages without any additional security mechanisms, as confirmed by the Theorem 6.30.

Q5: *Could existing programming languages be fixed or should new programming languages be developed?* The languages commonly used today have not necessarily been designed with the injection problem in mind. However, it is in general challenging to prove that a language is safe due to Corollary 6.27. For this reason, we believe it will be far easier to create new languages from the ground up, directly designed to avoid injection vulnerabilities. For simple languages, like network protocol, our recommendation is to rely on finite languages whose intent-security is decidable (Proposition 6.21) and to avoid infinite languages (notably infinite regular languages). We are convinced that methods must be found to better design grammars, for example, by searching for sufficient conditions for a grammar to be intent-secure. Indeed, safe grammars exist, as confirmed by the Proposition 6.9.

This theory highlights limitations in existing security approaches and could lead to new security strategies. The performances of today's IDS results either from a learning phase that identifies typical benign queries (in the case of behavioral IDS) or from the signatures of typical injections such as the infamous `' or '1'='1`. However, they rarely leverage the formal grammar that describes the language, causing non-zero false positive rate (i.e., some alerts don't correspond to attacks) and false negative rate (i.e., some attacks are not detected). Our study shows that the concept of developer intent could lead to better, provable protections.

Besides, knowing their intent, a developer could better test injections rather than relying on hazy pattern-matching. For example, they can use a parser to check if their intent derives the user input (Proposition 5.3). This strategy is also valid for several blanks. A protection system could automatically support this test if the developer's intent is known. It would behave similarly to prepared statements in SQL. However, in prepared statements, the developer is limited to some terminals while this system could support any intent, simple or complex, including non-terminals. Besides, such a system would be applicable to any context-free grammar and not only to SQL.

Finally, an intent analysis could be performed to ensure whether these tests are necessary (Theorems 5.12, 5.15 and 5.14) for most grammars used to date (regular, LR(0), and LR(k) in particular). Sadly, this test is not possible for linear and more complex grammars (Theorem 5.18). This last point is problematic because computing performance increases incentives for developers to use more general grammar parsers. For example, the lexical grammar of the Rust language is not context-free because of its raw strings[6], neither is the syntax of the Python language because of its semantic indentation.

Based on our results and the previous discussion, we propose a set of research directions that could both extend this present work and propose new security tools.

- An analysis of popular programming languages may identify some intents for which the language is intent-secure. One can distinguish data tokens (such as strings, identifiers, etc.) that can typically contain user data from control tokens (keywords such as `function`, `SELECT`, `<title>`, etc. and symbols such as `{`,`[`,`(`, etc.) that are typically written by the developer only and not filled by the user. Pragmatically, a query language does not have to be intent-secure for every symbol but only for data tokens.

[6]`https://github.com/rust-lang/rust/blob/0.12.0/src/grammar/raw-string-literal-ambiguity.md`

- At the moment, our theory is adapted to context-free grammars. However, most network protocol languages are regular with contextual fields (such as CRC values and message length). A focus on this class could bring new results for their risk analysis. Indeed, all contextual fields are not as useful against injections: adding a message length, for example, does not make the grammar classes much more complicated [67].
- Given the developer intent and the user input, it is easy to verify whether the injection is expected or not. This intent could be infered from unit tests, or by statically analyzing the templates of LR(k) grammars (if the intent length is 1). In both cases, it would be easy to verify user inputs with respect to the inferred developer intent.
- Even though it is difficult to prove the intent-security of established languages, one could provide sufficient and necessary conditions for grammars of future languages to be intent-secure, and enable a new generation of intent-secure query languages.

References

[1] J. Forristal, NT web technology vulnerabilities, accessed: 2022-06-08 (1998).
URL `http://phrack.org/issues/54/8.html`

[2] E. Poll, Langsec revisited: input security flaws of the second kind, in: 2018 IEEE Security and Privacy Workshops (SPW), IEEE, 2018, pp. 329–334.

[3] CWE, 2021 CWE top 25 most dangerous software errors, accessed: 2022-06-08 (2021).
URL `https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html`

[4] OWASP, The ten most critical web application security risks, accessed: 2022-06-08 (2017).
URL `https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf`

[5] OWASP, The ten most critical web application security risks, accessed: 2022-06-08 (2021).
URL `https://owasp.org/Top10/#welcome-to-the-owasp-top-10-2021`

[6] P. Bisht, P. Madhusudan, V. Venkatakrishnan, Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks, ACM Transactions on Information and System Security 13 (2) (2010) 14. `doi:https://doi.org/10.1145/1698750.1698754`.

[7] D. E. Denning, An intrusion-detection model, IEEE Transactions on software engineering SE-13 (2) (1987) 222–232.

[8] B. Miller, L. Fredriksen, B. So, An empirical study of the reliability of unix utilities, Communications of the ACM 33 (12) (1990) 32–44. `doi:10.1145/96267.96279`.

[9] A. Doupé, M. Cova, G. Vigna, Why johnny can't pentest: An analysis of black-box web vulnerability scanners, in: C. Kreibich, M. Jahnke (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment, Vol. 6201 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 111–131. `doi:10.1007/978-3-642-14215-4_7`.
URL `http://dx.doi.org/10.1007/978-3-642-14215-4_7`

[10] J. Fonseca, M. Vieira, H. Madeira, Testing and comparing web vulnerability scanning tools for sql injection and xss attacks, in: 13th Pacific Rim International Symposium on Dependable Computing, IEEE, Melbourne, Victoria, Australia, 2007, pp. 365–372. `doi:10.1109/PRDC.2007.55`.

[11] A. Dessiatnikoff, R. Akrout, E. Alata, M. Kaâniche, V. Nicomette, A clustering approach for web vulnerabilities detection, in: 17th Pacific Rim International Symposium on Dependable Computing, IEEE, Pasadena, CA, USA, 2011, pp. 194–203. `doi:10.1109/PRDC.2011.31`.
URL `http://dx.doi.org/10.1109/PRDC.2011.31`

[12] R. Akrout, E. Alata, M. Kaâniche, V. Nicomette, An automated black box approach for web vulnerability identification and attack scenario generation, Journal of the Brazilian Computer Society 20 (1) (2014) 4:1–4:16. `doi:10.1186/1678-4804-20-4`.
URL `http://dx.doi.org/10.1186/1678-4804-20-4`

[13] M.-T. Trinh, D.-H. Chu, J. Jaffar, S3: A symbolic string solver for vulnerability detection in web applications, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM, New York, NY, USA, 2014, pp. 1232–1243. `doi:10.1145/2660267.2660372`.
URL `http://doi.acm.org/10.1145/2660267.2660372`

[14] F. Yu, M. Alkhalaf, T. Bultan, O. H. Ibarra, Automata-based symbolic string analysis for vulnerability detection, Formal Methods in System Design 44 (1) (2014) 44–70. `doi:10.1007/s10703-013-0189-1`.
URL `https://doi.org/10.1007/s10703-013-0189-1`

[15] K. Elshazly, Y. Fouad, M. Saleh, A. Sewisy, A survey of sql injection attack detection and prevention, Journal of Computer and Communications 2 (08) (2014) 1. `doi:10.4236/jcc.2014.28001`.

[16] H. Dehariya, P. K. Shukla, M. Ahirwar, A survey on detection and prevention techniques of sql injection attacks, International Journal of Computer Applications 137 (5) (2016) 9–15. `doi:10.5120/ijca2016908672`.

[17] S. M. H. Chaki, M. M. Din, A survey on sql injection prevention methods, International Journal of Innovative Computing 9 (1) (2019) 47–54. doi:https://doi.org/10.11113/ijic.v9n1.224.

[18] G. Deepa, P. S. Thilagam, Securing web applications from injection and logic vulnerabilities, Information and Software Technology 74 (C) (2016) 160–180. doi:10.1016/j.infsof.2016.02.005.
URL http://dx.doi.org/10.1016/j.infsof.2016.02.005

[19] Z. Su, G. Wassermann, The essence of command injection attacks in web applications, SIGPLAN Not. 41 (1) (2006) 372–382. doi:10.1145/1111320.1111070.
URL https://doi.org/10.1145/1111320.1111070

[20] M. R. Islam, M. S. Islam, Z. Ahmed, A. Iqbal, R. Shahriyar, Automatic detection of nosql injection using supervised learning, in: V. Getov, J. Gaudiot, N. Yamai, S. Cimato, J. M. Chang, Y. Teranishi, J. Yang, H. V. Leong, H. Shahriar, M. Takemoto, D. Towey, H. Takakura, A. Elçi, S. Takeuchi, S. Puri (Eds.), 43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, Milwaukee, WI, USA, July 15-19, 2019, Volume 1, IEEE, 2019, pp. 760–769. doi:10.1109/COMPSAC.2019.00113.
URL https://doi.org/10.1109/COMPSAC.2019.00113

[21] B. Hou, Y. Shi, K. Qian, L. Tao, Towards analyzing mongodb nosql security and designing injection defense solution, in: 2017 ieee 3rd international conference on big data security on cloud (bigdatasecurity), ieee international conference on high performance and smart computing (hpsc), and ieee international conference on intelligent data and security (ids), 2017, pp. 90–95. doi:10.1109/BigDataSecurity.2017.29.

[22] M. Liu, B. Zhang, W. Chen, X. Zhang, A survey of exploitation and detection methods of xss vulnerabilities, IEEE Access 7 (2019) 182004–182016. doi:10.1109/ACCESS.2019.2960449.

[23] M. Baykara, S. Güçlü, Applications for detecting xss attacks on different web platforms, in: 2018 6th International Symposium on Digital Forensic and Security (ISDFS), 2018, pp. 1–6. doi:10.1109/ISDFS.2018.8355367.

[24] N. M. Sheykhkanloo, A learning-based neural network model for the detection and classification of sql injection attacks, Int. J. Cyber Warf. Terror. 7 (2) (2017) 16–41. doi:10.4018/IJCWT.2017040102.
URL https://doi.org/10.4018/IJCWT.2017040102

[25] R. Jahanshahi, A. Doupé, M. Egele, You shall not pass: Mitigating sql injection attacks on legacy web applications, in: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, 2020, pp. 445–457.

[26] S. Son, K. S. McKinley, V. Shmatikov, Diglossia: detecting code injection attacks with precision and efficiency, in: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, 2013, pp. 1181–1192.

[27] W. G. Halfond, A. Orso, Amnesia: analysis and monitoring for neutralizing sql-injection attacks, in: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ASE '05, Association for Computing Machinery, New York, NY, USA, 2005, pp. 174–183. doi:10.1145/1101908.1101935.
URL https://doi.org/10.1145/1101908.1101935

[28] W. Halfond, A. Orso, P. Manolios, Wasp: Protecting web applications using positive tainting and syntax-aware evaluation, IEEE transactions on Software Engineering 34 (1) (2008) 65–81.

[29] O. C. Abikoye, A. Abubakar, A. H. Dokoro, O. N. Akande, A. A. Kayode, A novel technique to prevent sql injection and cross-site scripting attacks using knuth-morris-pratt string match algorithm, EURASIP Journal on Information Security 2020 (1) (2020) 1–14.

[30] A. Pramod, A. Ghosh, A. Mohan, M. Shrivastava, R. Shettar, Sqli detection system for a safer web application, in: 2015 IEEE International Advance Computing Conference (IACC), IEEE, 2015, pp. 237–240.

[31] M. Hoschele, A. Zeller, Mining input grammars with autogram, in: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), IEEE, 2017, pp. 31–34.

[32] H. Gu, J. Zhang, T. Liu, M. Hu, J. Zhou, T. Wei, M. Chen, Diava: a traffic-based framework for detection of sql injection attacks and vulnerability analysis of leaked data, IEEE Transactions on Reliability 69 (1) (2019) 188–202.

[33] K. Mookhey, N. Burghate, Detection of sql injection and cross-site scripting attacks, Symantec SecurityFocus.

[34] M. Wan, K. Liu, An improved eliminating sql injection attacks based regular expressions matching, in: 2012 International Conference on Control Engineering and Communication Technology, IEEE, 2012, pp. 210–212.

[35] A. Sravanthi, K. J. Devi, K. S. Reddy, A. Indira, V. S. Kumar, Detecting sql injections from web applications, International Journal Of Engineering Science & Advanced Technology [IJESAT] 2 (3).

[36] B. Kranthikumar, R. L. Velusamy, Sql injection detection using regex classifier, Journal of Xi'an University of Architecture & Technology 12 (6) (2020) 800–809.

[37] S. Steiner, D. C. de Leon, J. Alves-Foss, A structured analysis of SQL injection runtime mitigation techniques, in: T. Bui (Ed.), 50th Hawaii International Conference on System Sciences, HICSS 2017, Hilton Waikoloa Village, Hawaii, USA, January 4-7, 2017, ScholarSpace / AIS Electronic Library (AISeL), 2017, pp. 1–9.
URL http://hdl.handle.net/10125/41505

[38] A. Liu, Y. Yuan, D. Wijesekera, A. Stavrou, Sqlprob: a proxy-based architecture towards preventing sql injection attacks, in: Proceedings of the 2009 ACM symposium on Applied Computing, SAC '09, Association for Computing Machinery, New York, NY, USA, 2009, pp. 2054–2061. doi:10.1145/1529282.1529737.
URL https://doi.org/10.1145/1529282.1529737

[39] S. W. Boyd, A. D. Keromytis, Sqlrand: Preventing sql injection attacks, in: M. Jakobsson, M. Yung, J. Zhou (Eds.), International Conference on Applied Cryptography and Network Security, Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 292–302.

[40] D. Ray, J. Ligatti, Defining code-injection attacks, in: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'12, ACM, New York, NY, USA, 2012, pp. 179–190. doi:10.1145/2103656.2103678.

URL https://doi.org/10.1145/2103656.2103678
[41] R. Hansen, M. Patterson, Stopping injection attacks with computational theory (2005).
URL https://www.blackhat.com/presentations/bh-usa-05/bh-us-05-hansen.pdf
[42] R. J. Hansen, M. L. Patterson, Guns and butter: Towards formal axioms of input validation (2005).
[43] L. Sassaman, M. L. Patterson, S. Bratus, M. E. Locasto, Security applications of formal language theory, IEEE Systems Journal 7 (3) (2013) 489–500.
[44] T. Bieschke, L. Hermerschmidt, B. Rumpe, P. Stanchev, Eliminating input-based attacks by deriving automated encoders and decoders from context-free grammars, in: 2017 IEEE Security and Privacy Workshops (SPW), IEEE, 2017, pp. 93–101.
[45] J. Bangert, N. Zeldovich, Nail: A practical tool for parsing and generating data formats, in: 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14), 2014, pp. 615–628.
[46] F. Momot, S. Bratus, S. M. Hallberg, M. L. Patterson, The seven turrets of babel: A taxonomy of langsec errors and how to expunge them, in: 2016 IEEE Cybersecurity Development (SecDev), IEEE, 2016, pp. 45–52.
[47] T. Dullien, Weird machines, exploitability, and provable unexploitability, IEEE Transactions on Emerging Topics in Computing 8 (2) (2017) 391–403.
[48] N. Chomsky, On certain formal properties of grammars, Information and control 2 (2) (1959) 137–167. doi:10.1016/S0019-9958(59)90362-6.
URL http://www.sciencedirect.com/science/article/pii/S0019995859903626
[49] P. S. Landweber, Decision problems of phrase-structure grammars, Electronic Computers, IEEE Transactions on EC-13 (4) (1964) 354–362.
[50] J. E. Hopcroft, J. D. Ullman, Formal Languages and Their Relation to Automata, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1969. doi:10.1137/1012073.
[51] S. Ginsburg, E. H. Spanier, Quotients of context-free languages, Journal of the ACM 10 (4) (1963) 487–492. doi:10.1145/321186.321191.
[52] S. Ginsburg, S. A. Greibach, Deterministic context free languages, Information and Control 9 (6) (1966) 620–648. doi:10.1016/S0019-9958(66)80019-0.
URL https://doi.org/10.1016/S0019-9958(66)80019-0
[53] D. E. Knuth, On the translation of languages from left to right, Information and Control 8 (6) (1965) 607–639. doi:https://doi.org/10.1016/S0019-9958(65)90426-2.
URL http://www.sciencedirect.com/science/article/pii/S0019995865904262
[54] M. M. Geller, M. A. Harrison, On lr(k) grammars and languages, Theoretical Computer Science 4 (3) (1977) 245–276. doi:10.1016/0304-3975(77)90013-5.
[55] R. Alur, P. Madhusudan, Visibly pushdown languages, in: Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing, STOC '04, Association for Computing Machinery, New York, NY, USA, 2004, p. 202–211. doi:10.1145/1007352.1007390.
URL https://doi.org/10.1145/1007352.1007390
[56] G. Rozenberg, Arto salomaa, editors. 1997. handbook of formal languages, volume 1 word, language, grammar.
[57] P. R. Asveld, A. Nijholt, The inclusion problem for some subclasses of context-free languages, Theoretical computer science 230 (1-2) (2000) 247–256.
[58] D. J. Rosenkrantz, R. E. Stearns, Properties of deterministic top-down grammars, Information and Control 17 (3) (1970) 226–256.
[59] T. Kasami, An efficient recognition and syntax analysis algorithm for context-free languages, Tech. Rep. R-257, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign (Mar. 1966).
[60] G. Sénizergues, The equivalence problem for deterministic pushdown automata is decidable, in: P. Degano, R. Gorrieri, A. Marchetti-Spaccamela (Eds.), International Colloquium on Automata, Languages, and Programming, Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 671–681. doi:https://doi.org/10.1007/3-540-63165-8_221.
[61] A. Yehudai, The decidability of equivalence for a family of linear grammars, Information and Control 47 (2) (1980) 122–136.
[62] B. S. Baker, R. V. Book, Reversal-bounded multipushdown machines, Journal of Computer and System Sciences 8 (3) (1974) 315–332.
[63] J. Berstel, L. Boasson, Formal properties of xml grammars and languages, Acta Informatica 38 (9) (2002) 649–671.
[64] M. Latteux, P. Turakainen, On characterizations of recursively enumerable languages, Acta Informatica 28 (2) (1990) 179–186. doi:10.1007/BF01237236.
URL http://dx.doi.org/10.1007/BF01237236
[65] B. Beizer, Software testing techniques, Dreamtech Press, 2003.
[66] G. Hoglund, G. McGraw, Exploiting software: How to break code, Pearson Education India, 2004.
[67] S. Lucks, N. M. Grosch, J. König, Taming the length field in binary data: calc-regular languages, in: 2017 IEEE Security and Privacy Workshops (SPW), IEEE, 2017, pp. 66–79.