Towards programming languages free of injection-based vulnerabilities by design

Eric Alata
LAAS-CNRS, INSA
Toulouse, France
eric.alata@laas.fr

Pierre-François Gimenez
Univ. Rennes, Inria, IRISA
Rennes, France
pierre-francois.gimenez@inria.fr

***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. From these definitions, we show that new design patterns can help create programming languages that are secure by design.**

***Index Terms*—Programming Languages, Formal Languages, Security**

The authors contributed equally to this work.

I. 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. A whole class of attacks aims 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. They can produce dramatic effects depending on the type of service, and their persistent vulnerability is generally attributed to a lack of developer awareness or bad practices [1]. In 2020, CWE (Common Weakness Enumeration) [2] 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 [3] and as the third most dangerous one in 2021 [4]. 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 formats such 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 [5], we consider that the developer has 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, our contributions are:

- two new security properties: intent-equivalence and

[1]HTTP: CVE-2019-13143, SQL: CVE-2019-1010259 and CVE-2019-14900; 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; Bash: CVE-2014-6271

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

[4]URL: CVE-2022-0391, JSON: CVE-2018-14010

intent-security;
- decidability of such properties for various grammar classes;
- grammar design patterns that can lead to languages free of injection-based vulnerabilities.

The paper is structured as follows. Section II presents some fundamental background concepts of language theory. Our fundamental assumptions and their technical implications are discussed in Section III. The intent-equivalence property is defined and studied in Section IV. The intent-security property is defined and analyzed in Section V. Section VI presents strategy to design languages that are secure by design. Section VII discusses related work and positions our main contributions. Finally, Section VIII summarizes our major results and discusses future directions.

II. Background

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

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 $B \backslash A$, is the set $\{v \in T^* \mid \exists u \in T^* : uv \in A \wedge u \in B\}$. Similarly, the right quotient A/B is $\{u \in T^* \mid \exists v \in T^* : uv \in A \wedge v \in B\}$. We will denote $p \backslash A$ the left quotient of A by the language containing solely the word $p \in T^*$ and A/s the right quotient of A by the language containing solely the word $s \in T^*$.

The hierarchy of Noam Chomsky classifies grammars into several types [6], according to the restrictions on the form of the rules. This classification is briefly described in Table I. 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 [7]–[9]. 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 I: Chomsky hierarchy

For context-free languages, one can represent a derivation in the form of a rooted tree, called *parse 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 parse 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 [10] 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.

Operation	REG	LL(1)	LR(0)	DCFL	CFL
Concatenation	✓	✗	✓	✗	✓
Union	✓	✗	✗	✗	✓
Complement	✓	✗	✗	✓	✗
Intersection with reg. set	✓	✗	✗	✓	✓
Inverse homomorphism	✓	✗	✗	✓	✓

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

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

TABLE III: Decidability (D) and undecidability (U) of various problems on languages classes.

Table II summarizes the closures and non-closures properties of operations for classic grammar classes [8], [11]. 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 III summarizes the decidability of various problems for the same grammar classes [12], [13]: membership, equivalence, inclusion and emptiness.

III. Malicious injections modelization

Injections vulnerabilities typically stem from the integration of untrusted user input to build queries. In this paper, we use the term query to denote the complete string that includes the part written by the developer and the part that stems from the user input. 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.

Let us illustrate new concepts we introduce with the LDAP language, because it has a simple grammar, is widely used and can be targeted by injection-based attacks. A simplified grammar G_{LDAP} can be expressed with the following rules:

$$S \to \texttt{(!}S\texttt{)} \quad S \to \texttt{(s=s)} \quad S \to \texttt{(\&}L\texttt{)}$$
$$S \to \texttt{(|}L\texttt{)} \quad L \to S \quad L \to LS$$

where S is the axiom, L is a non-terminal denoting a list, and s is a token that can match any alphanumerical string. Semantically, `!` is the negation, `&` is the logical conjunction ("AND") and `|` is the logical disjunction ("OR"). This grammar is LR(0) and is not regular.

For the sake of illustration, assume the developer maintains directory information services that can be searched with LDAP, a widely used protocol, especially for user authentication. They design the following template to check if a username and a password exist:

`(&(uid=␣)(passwd=␣))`

This template has two blanks, denoted ␣, where user inputs can be injected. The developer expect the user to input one string for each blank so, intuitively, any user input that is not a string should be considered malicious. Sometimes, the developer can expect a more complex user input, like a list of comparisons, or a Boolean formula. We assume that legitimate injections comply with the developer's intent which can be modelized as a symbol of the grammar, or a sequence of symbols. In the previous template, the intent of the developer can be modelized as the terminal `s`.

We justify this choice as follow. A malicious injection is an injection that modifies the semantics of the query. We cannot use this definition as formal language theory does not model semantics. However, the semantics is handled by parsers 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. 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 are formally defined in the next sections.

IV. Intent-equivalence

A. Definitions

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 in the appendix. 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 IV.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:

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

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 IV.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. 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.

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.

Example IV.3. *Consider the following LDAP template t:*

`(&(uid=␣)(passwd=1234))`

Let us assume the intent of the developer is the terminal `s`, i.e., the user is expected to enter a string. Since `s` is a terminal, the only expected injection is `s`, i.e., $E(G_{LDAP}, \texttt{s}) = \{\texttt{s}\}$. However, the injection `foo)(loc=bar` lead to the syntactically valid query:

`(&(uid=foo)(loc=bar)(passwd=1234))`

So `foo)(loc=bar` $\in F(L(G_{LDAP}), t)$ and `foo)(loc=bar` $\notin E(G_{LDAP}, \texttt{s})$. Therefore, this template can accept injections that do not comply with the intent of the developer: this template is therefore vulnerable.

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 IV.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 that, 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 $E(G, \iota) \subseteq F(L(G), (p, s))$.

Example IV.5. *Let us consider the grammar $G = (\{a, b, c, d\}, \{S\}, R, S)$ where $R = \{S \to aSb; S \to cd\}$, and the following template:*

$$aa\_cdbbb$$

Here, the only injection that lead to a syntactically correct sentence is "a". Therefore, this template is intent-equivalent to "a".

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 definitions of Section IV for templates with m blanks and intentions that are tuples. As a reminder, we denote tuples in boldface.

A template is a sequence of terminals with some blanks: $t_1 \__1 t_2 \__2 t_3 \dots t_m \__m t_{m+1}$. 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 IV.6 (Template with m blanks)**.** Let $m \geq 1$. A template with m blanks is a sequence $\mathbf{t} = (t_1, \dots, t_{m+1}) \in T^* \times (T^+)^{m-1} \times T^*$.

The template `(&(uid=␣)(passwd=␣))` presented earlier is a typical example of a template with two blanks used for authentication.

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 IV.7 (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.

We extend the other definitions to multiple blanks in Appendix A.

Example IV.8. *Reusing the grammar defined in Example IV.5, the template aa_cd_bb is not intent-equivalent to (a, b): indeed, the attacker can inject aa in the first blank and bb in the second blank. In fact, there is not intent that can derive exactly the set of possible injections, even if both aa_cdbbb and $aaacd\_bb$ are independently intent-equivalent to a and b respectively.*

B. Properties

It can be of interest to assess the decidability of the intent-equivalence of a template $\mathbf{t}$ to some fixed intent $\boldsymbol{\iota}$ for various classes of grammar. This problem mainly concerns the static analysis of the source code, where the tool has access to the template. We skip the study of finite languages whose intent-security is trivially decidable. The first class we study is those of the regular grammars.

Theorem IV.9. *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:

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

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 us 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, $L_2 = \mathbf{t} \odot (\#_1 T^* \overline{\#}_1, \#_2 T^* \overline{\#}_2, \ldots, \#_m T^* \overline{\#}_m)$. 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, $f_{\mathbf{t}}(E(G,\boldsymbol{\iota})) = \{\mathbf{t} \odot (\#_1 w_1 \overline{\#}_1, \#_2 w_2 \overline{\#}_2, \ldots, \#_m w_m \overline{\#}_m) \mid \forall i, w_i \in E(G,\iota_i)\}$. 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 [8]). ∎

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. While $p\backslash L(G)/s$ 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 IV.10. *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.*

The proof of this lemma, along with others, is available in Appendix B.

This lemma allows us to adapt the proof of Theorem IV.9 and to conclude the decidability of the intent-equivalence for LR(k) grammar when the intent length is 1.

Theorem IV.11. *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 IV.9. 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 IV.10) and because $f_{\mathbf{t}}(F(L,t))$ is a marked concatenation of LR(k) languages, $f_{\mathbf{t}}(F(L,t))$ is deterministic [14]. So $F(L,\mathbf{t})$ and $E(G,\boldsymbol{\iota})$ are deterministic. Finally, the equivalence of deterministic languages is decidable [15]. We can conclude that it is decidable whether $\mathbf{t}$ is intent-equivalent to $\boldsymbol{\iota}$. ∎

This theorem is limited to $|\iota| = 1$ 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 IV.12. *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 IV.11. 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 [14], $E(G,\iota_i)$ is an $LR(0)$ language. The rest of the proof is similar. ∎

Example IV.13. *G_{LDAP} is LR(0) so the intent-equivalence of any of its templates could be automatically assessed. For example:*

`(!(uid=foo)`␣

is intent-equivalent to `)` *and*

`(&(uid=`␣`)(passwd=1234))`

is not intent-equivalent to `s` *(as discussed in Example IV.3).*

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.

Theorem IV.14. *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.*

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 IV.15. *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 IV.9. 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 IV.9 and IV.11 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, which encompass many real-world 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 context-free grammars due to Theorem IV.14. The question of the decidability of the intent-equivalence for any intent is still open for deterministic context-free grammars.

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

V. Intent-security

A. Definitions

The goal of this subsection is to propose a property that guarantees that a grammar is safe. Consider the following case: a developer wants to write an LDAP query where the user provides the value for some field. 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.

Definition V.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 V.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 V.3. *By relying on expected injections, we can propose an alternative and equivalent definition of unexpected injections: $\delta I(G, \iota) = I(G, \iota) - E(G, \iota)$*

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 V.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 V.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: G is intent-secure for ι

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

$\iff I(G, \iota) \subseteq E(G, \iota)$

$\iff \iota \Rightarrow^* w$ for all $w, p, s \in T^*$
such that $pws \in L(G)$ and $S \Rightarrow^* p\iota s$

$\iff (p, s)$ is intent-equivalent to ι for all $(p, s) \in T^*$
such that $S \Rightarrow^* p\iota s$

∎

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. The extensions of these definitions to multiple blanks are available in Appendix A.

Example V.6. *A fair question is: do such grammars exist? This will be the subject of Section VI. Small finite grammar can trivially be intent-secure, but let us show that the grammar in Example IV.5, that describes an infinite language, is in fact intent-secure for intent length 1.*

A simple proof by induction shows that $L(G) = \{a^n c d b^n \mid n \geq 0\}$. 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 $a^m a a^r cdb^n$ 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 $I(G, S) = \bigcup_{n \geq 0} a^n \backslash L / b^n = \{a^m cdb^m \mid m \geq 0\} = E(G, S)$. So $\delta I(G, S) = \varnothing$.*

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

Definition V.7 (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.

B. Properties

Let us first 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 [16] 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 V.8. *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 $\iota \in \Delta^+$.*

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 (cf. Example B.1 in Appendix B).

We can also derive a monotonic property on languages (and not grammars), provided we restrict the analysis to terminal intents. Trivially, if a language is inherently intent-insecure, then any superset of this language is also inherently intent-insecure, as implied by the following proposition.

Proposition V.9. *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 $\delta I(G_1, \iota) \subseteq \delta I(G_2, \iota)$.*

The rest of the subsection is dedicated to analyzing the decidability of the intent-security for classical classes of grammars: 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.

The intent-security is trivially decidable for finite languages, because all considered sets are finite. However, such a result cannot be obtained for infinite regular languages. In fact, these languages are inherently insecure due to the pumping lemma.

Theorem V.10. *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 $|y| \geq 1$, $|xy| \leq l$, and $\forall k \geq 0,\ xy^k z \in L(G)$. As $|y| \geq 1$, $|y^n| \geq n$. Let w, w' be two words 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 L is inherently intent-insecure for intent length n no matter what grammar is used to describe it. ∎

This theorem may appear of little interest, as regular languages are rarely used in programming languages. However, due to the Proposition V.9, we can conclude that any language that includes an infinite regular language is also inherently intent-insecure.

Corollary V.11. *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 V.9 and Theorem V.10. ∎

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, it explains their vulnerability to injection-based attacks.

Example V.12. *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. This is also the case for piggy-backed SQL injections that rely on the `(<Query> ;)* <Query>` regular sublanguage and for UNION-based SQL injections that rely on the `(<Select Query> UNION)* <Select Query>` regular sublanguage.

In fact, this is also the case for OS command injection `(<Command> ;)* <Command>`, for SMTP "To" and "From" injection `(<Email> %0A cc:)* <Email>`, for LDAP injection (cf. Example IV.3), and many more.

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 V.13. *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 unexpected injections in k are $\delta I(G, k) = \{\hat{v}a^{2^i} \mid i \geq 0\}$, which is context-sensitive and not context-free.

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, for more complex grammars, the intent-security is undecidable.

Theorem V.14. *Let $n \geq 1$. A language L is recursively enumerable if and only if there exist 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 V.15. *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. ■

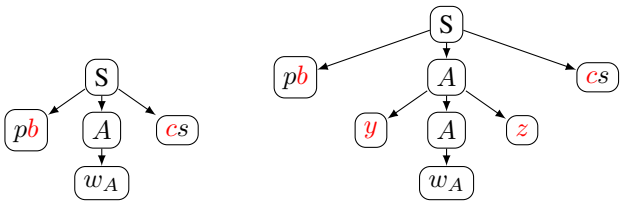
If the intent-security is undecidable for the class of LR(0) grammars, it is also undecidable for the classes that include LR(0) grammars, such as LR(k) grammars, deterministic grammars and context-free grammars.

These results show there is no automatic way of proving that some LR(0) 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 IV) is a more promising approach.

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

Theorem V.16. *Infinite context-free languages are inherently intent-insecure for at least two blanks.*

Proof: This result is an application of the pumping lemma for context-free languages. Let G be a context-free grammar that describes an infinite language. If G describes a regular language, then according to Theorem V.10 it is not intent-secure with a single blank, so it cannot be intent-secure with two blanks.



(a) Simplified version of the parse tree of pbw_Acs

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

Fig. 1: Illustration of the proof of Theorem V.16. Injections are in red.

Let us assume that G describes a non-regular language. As remarked by Chomsky in [6], 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 us 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 us 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 us 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 1.

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. ■

Example V.17. *The injection vulnerability stemming from the context-free pumping lemma is less prevalent because it requires two injection points. However, consider again this LDAP template:*

`(&(uid=␣)(passwd=␣))`

We can safely assume that the developer expects a string s into each injection point, leading to the parse tree presented in Figure 2.

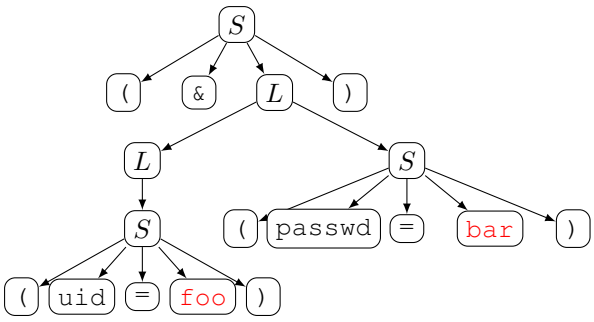


Fig. 2: Parse tree for expected injections.

If the injection in the first blank is `admin)(!(&(1=0` *and the injection in the second blank is* `text))`*, then the final query is:*

`(&(uid=admin)(!(&(1=0)(passwd=text)))`

where the injections are in red to improve readability. This classical LDAP injection bypasses the password check by pumping the S nonterminal, as shown in Figure 3. Sadly, there is no way to avoid this kind of vulnerabilities without leaving the context-free language class.

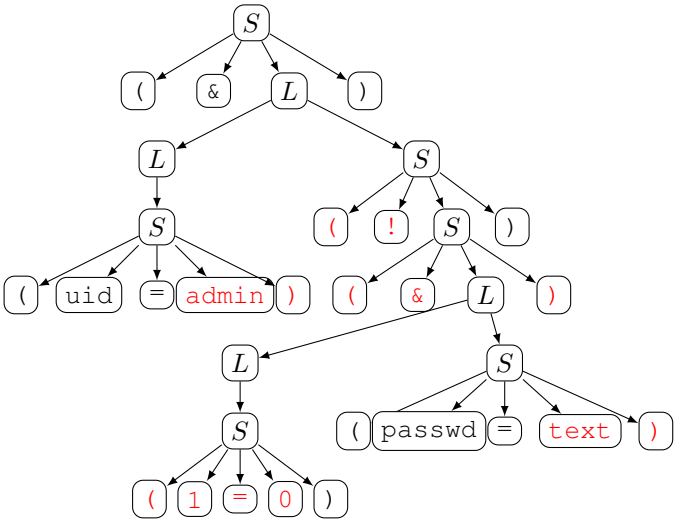


Fig. 3: Parse tree for the injection attack.

While our study pinpoints universal injection vulnerabilities in infinite regular languages and context-free languages due to the pumping lemmas, there exist of course injections that do not rely on the pumping lemmas. For example, the SQL template `SELECT * FROM products WHERE id=␣` could expect a number but receive a string. However, a wide variety of injections attacks do in fact exploit these pumping lemmas.

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.

VI. Intent-security by design

Theorem V.15 shows that there is no general algorithm that can prove that a language is intent-secure, which can give the feeling that the notion of intent-secure language has no practical interest and that injection vulnerabilities are unavoidable. In this section, we show that practical intent-secure languages are possible and we propose design patterns that can lead to intent-secure languages by design.

Theorem VI.1. *Let $G = (T, N, R, S)$ be a context-free formal grammar. Let us denote G_A the grammar (T, N, R, A) where $A \in N$. If*

- *G is $LL(1)$,*
- *G is $RR(1)$,*
- *G is epsilon-free,*
- *$L(G_A)$ is bifix-free (prefix-free and suffix-free),$\forall A \in N$,*
- *For all $A \in N$, if there exists $B \in \Delta$ and $\alpha \in \Delta^*$ such that $A \to B$ and $A \to \alpha$, then $\alpha = B$,*

then G is intent-secure for any intent $\iota \in \Delta$.

The intuition behind this proposition is that the intent can be inferred from the left part of the query (because it is $LL(1)$ and the right part of the query (because it is $RR(1)$). The last two condition are in fact necessary: the bifix-free condition is necessary to avoid attacks that expand on the left or the right of the expected user input and the last condition is necessary to identify the correct intent.

From this result, we can derive a simple design pattern: by adding opening (resp. closing) delimiters o (resp. c) for each rule r, we can ensure the previous conditions are met.

Proposition VI.2. *Let $G = (T, N, R, S)$ be a context-free formal grammar. If for all rules r, there exist $A \in N$, $\alpha \in \Delta^*$, $o_r \in T$ and $c_r \in T$ such that $r : A \to o_r \alpha c_r$, and o_r and c_r only appear once in the grammar, then G is intent-secure for any intent $\iota \in \Delta$.*

Proof: We show that G verifies the condition of the previous theorem. The parsing is guided by the first symbol of each rule: since it is unique, only one rule can explain its occurrence so G is $LL(1)$. The same reasoning can show that G is $RR(1)$. Each rule produces at least two terminals (o_r and c_r), so there are no epsilon-production. The presence of these symbols makes the grammar bifix-free, and it is still bifix-free if the axiom is changed. Each rule contains at least

two symbols (o_r and c_r). Since o_r and c_r are unique, the last condition is also verified. So G is intent-secure for any intent $\iota \in \Delta$. ∎

Example VI.3. *This proposition can be applied to the grammar from Example V.6 to (re)prove that it is intent-secure for any intent $\iota \in \Delta$: indeed, its two rules $S \to aSb$ and $S \to cd$ each starts and ends with a unique symbol.*

Let us illustrate this design pattern on a simple example: a list. Lists in programming languages are typically written as $e_1, e_2, e_3, \ldots, e_n$. We discussed lists in Section V as the typically regular expression pattern that can be trivially injected. Such a list can be easily expressed with the following rules:

$$L \to e, L \quad L \to e$$

where e is an element of the list.

We can modify these rules as suggested in Prop. VI.2, by adding unique symbols at the start and the end of each rule:

$$L \to [e, L] \quad L \to <e>$$

where we assume that [,],< and > do not appear elsewhere in the language. The added delimiters (rendered in red for clarity) transform the original list grammar that was regular to a non-regular, context-free language that cannot be injected with only one blank: Prop. VI.2 proves that this modified list grammar is intent-secure for any intent $\iota \in \Delta$.

We encourage the reader to try to attack the following templates:

```
<__>
[1, [2, [__, <4>]]]
```

The attacker cannot modify the number of elements due to the new delimiters [,],< and >. These templates are still vulnerable if there are two injection points, as are all context-free languages.

Example VI.4. *Our theoretical work does not differentiate the parser from the lexer. In practice, however, the attacker could modify the template written by the developer by starting a comment. Assume for example that the previous language L allows single-line comments to start with `--`. Then, the attacker could for example inject `3, [4, <5>]]]]--` inside the second template, leading to:*

```
[1, [2, [3, [4, <5>]]]]--, <4>]]
```

which, after removal of the comment by the lexer, leads to:

```
[1, [2, [3, [4, <5>]]]]
```

which is a successful injection attack. This is not the case for comments with a mandatory end delimiter (e.g., comments with `/*` and `*/`) for single-blank injections. Remark that LDAP search filters do not allow comments.

We propose another design pattern that requires fewer new delimiters but assumes the intent of the developer is not a delimiter. This assumption is reasonable for many languages since delimiters are mostly useful for easier parsing while users typically input data symbols such as strings, values, etc.

Proposition VI.5. *Let $G = (T, N, R, S)$ be a context-free formal grammar. If for all rules r, there exist $A \in N$, $\alpha \in \Delta^*$, $o_r \in T$ and $c \in T$ such that $r : A \to o_r \alpha c$, o_r only appear once in the grammar and c only appear as the last element of right-hand parts of rules, then G is intent-secure for any intent $\iota \in \Delta - (\{o_r \mid r \in R\} \cup \{c\})$.*

As an example, let us protect the LDAP grammar by assuming the intent of the developer will only be `s`. In that case, templates such as `(uid=foo__` (intent: `)`) or `(__(uid=foo)(passwd=bar))` (intent: `&` or `|`) are forbidden. Applying Th. VI.5, we added new delimiters (rendered in red for clarity) to some rules:

$$S \to (!S) \quad S \to \{s{=}s\} \quad S \to (\&L)$$
$$S \to (|L) \quad L \to <S> \quad L \to [LS]$$

Each rule starts with a unique delimiter: `(!`, `{`, `(&`, `(|`, `<`, or `[`, so the theorem indeed applies. Therefore, the LDAP query:

```
(&(uid=foo)(passwd=__))
```

can be rewritten into (the modified symbols are in red):

```
(&[<{uid=foo}>{passwd=__}])
```

This template is intent-equivalent to `s`, meaning that only string can be input by the user. In fact, any template where the user should enter a string `s` will be secure against injection-based attacks, so the grammar itself is secure.

Such design patterns could be used to create new, secure programming languages, by effectively proving smaller part of the language and using operations that are closed for intent-secure grammars to merge them.

Proposition VI.6. *Let $G_1 = (T, N_1, R_1, S_1)$ and $G_2 = (T, N_2, R_2, S_2)$ be two context-free formal grammars.*

- *If G_1 and G_2 are intent-secure for any intent $\iota \in \Delta$, then the concatenated grammar $G_3 = (T, N_1 \cup N_2, R_1 \cup R_2 \cup \{S \to S_1 S_2\}, S)$ is intent-secure for any intent $\iota \in \Delta$.*
- *If G_1 is a subgrammar of G_2, i.e., if S_1 is reachable from S_2, and if G_2 is intent-secure, then G_1 is intent-secure.*
- *Let L be any language. For any $n \geq 1$, if G_1 is intent-secure for all $\iota \in T^n$, then so is $L(G_1) \cap L$.*
- *If G_1 is intent-secure for any intent $\iota \in \Delta$, then its reversal G_1^r is also intent-secure for any intent $\iota \in \Delta$.*

Proof sketch: For the concatenation: because G_1 and G_2 are intent-secure, they are also bifix-free, and so their concatenation is unambiguous. So, for any template, we can reduce the problem of intent-security to the intent-security of either G_1 or G_2. For the subgrammar: trivial from the definition of intent-security. The third result is simply the contraposition of V.9. ∎

However, intent-secure grammars are not closed for union. For example, $L_1 = \{aa\}$ and $L_2 = \{ab\}$ have intent-secure grammars but $L_1 \cup L_2$ does not: $b \in \delta I(L_1 \cup L_2, a)$.

Finally, we would like to point out a hopeful result: there probably exist context-sensitive languages that are not inherently intent-insecure for several blanks. Consider any context-free language L and the language L'_k defined as:

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

where $\#$ is a new symbol not present in L. L'_k is a variant of the copy language known to be context-sensitive but not context-free [8]. 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. 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. This result is not a formalized proposition as our definitions are not adapted to non-terminal intents for context-sensitive grammars.

This section shows the potential practical impact of our theoretical framework, and we believe that tighter sufficient conditions for intent-secure languages can be found, as well as more user-friendly design patterns.

VII. Related work

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. Let us examine the scope and limitations of the proposed definitions.

The authors of [17] 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 [5], 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.

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. Our work seek to expand the theoretical work on injection-based vulnerabilities previously initiated by [5], [17], [18]. 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.

In 2005, [19] used formal language results, and notably undecidability theorems, to analyze input validation and propose multiple observations and advice to developers. In [16], 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. These contributions, alongside [1], [20], are complementary to our work. Their work tackle 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. We consider that injection-based vulnerabilities have also a source in the formal grammar itself, and this is why we focus to its in-depth study.

The study of secure-by-design programming languages is not new. For example, the popular Rust language is memory-safe, protecting programs against memory-based vulnerabilities such as buffer overflow or pointers' use-after-free. The language Wyvern [21] proposes to embed in a single language SQL and HTML so the type system can verify if the user input matches the expected type. Sadly, this line of work did not transfer to the industry. Wyvern protection cannot extend over serialization an deserialization, so it cannot prevent an injection if the concatenation is performed by another system. On the other hand, Wyvern do not require to modify the grammar while we do. Therefore, we consider our work to be complementary to Wyvern.

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 [22]. 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 may receive 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 client's program (i.e., the path in the intended finite state machine). Injection attacks are challenging to study because they may happen at the interface of two systems with no weird states.

VIII. 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 in the first place. Furthermore, to the best of our knowledge, no research has been done on protections embedded in the language itself that could complement protections targeting implementation issues. 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. The decidability results of these two properties are summarized in Table IV and V.

	m blanks, $m \geq 1$		
	$\boldsymbol{\iota} \in (\Delta)^m$	$\boldsymbol{\iota} \in (\Delta^+)^m$	$\boldsymbol{\iota} \in (T^+)^m$
REG	D (Th. IV.9)	D (Th. IV.9)	D (Th. IV.15)
LR(0)	D (Th. IV.11)	D (Th. IV.12)	D (Th. IV.15)
LR(k)	D (Th. IV.11)	?	D (Th. IV.15)
CFG	U (Co. IV.14)	U (Co. IV.14)	D (Th. IV.15)

TABLE IV: Intent-equivalence property. D: decidable. U: undecidable. ?: unknown

	One blank	Two or more blanks
Finite	D	D
Non-finite REG	F (Th. V.10)	F (Th. V.10)
CFG with infinite regular sublanguage	F (Co. V.11)	F (Th. V.16)
Non-finite LR(0)	U (Co. V.15)	F (Th. V.16)

TABLE V: Intent-security property. F: always false. D: decidable. U: undecidable.

Our framework also allows to nuance some popular beliefs about injection-based vulnerabilities. Injection-based 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 [23]: "data validation is the first line of defense against a hostile world". However, Theorems V.14 and Corollary V.15 show that malicious injections may be very hard to detect. So, even if data validation is indeed a necessary line of defense, it cannot not sufficient. Injection vulnerabilities are embedded in the language itself and one cannot rely only on the developer's skills to avoid them. If one uses an intent-insecure grammar, additional security mechanisms, such as filtering, encoding, input sanitization, static analysis, or intrusion detection, are essential.

Besides, it is generally believed that simpler languages are less prone to being attacked, as explained in [24]: "a complex computational system is an engine for executing malicious computer programs delivered in the form of crafted input". Simple languages are very useful as the intent-security and intent-equivalence of finite languages is decidable. 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 V.10), while context-free languages may be intent-secure. 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.

To avoid injection-based vulnerabilities, we proposed several syntactical design patterns. While we consider them too cumbersome for mainstream programming languages, they can be applied successfully to Domain-Specific Languages (DSL) in domains where security properties are paramount. Besides, we are certain that more user-friendly design patterns can be proposed by the community.

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. First, 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`, 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. Second, we proved some decidable intent-equivalence problems but we didn't provide effective algorithms. Notably, an algorithm to decide the intent-equivalence (or lack of) of a query could be very useful for static analysis. Third, since infinite context-free grammars are vulnerable to multiple injections and their intent-security is undecidable, is it possible to transform any context-free grammar into a similar intent-secure context-sensitive grammar? We believe that indexed grammars are particularly suitable for this task, since they are context-sensitive and yet have efficient parsers [25]. Finally, at the moment, our theory is adapted to context-free grammars. However, most network protocol languages are regular with contextual fields (such as checksum 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 [26].

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and Erik Poll for their invaluable feedback on the article.

REFERENCES

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

[2] CWE, "2021 CWE top 25 most dangerous software errors," 2021, accessed: 2022-06-08. [Online]. Available: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

[3] OWASP, "The ten most critical web application security risks," 2017, accessed: 2022-06-08. [Online]. Available: https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf

[4] ——, "The ten most critical web application security risks," 2021, accessed: 2022-06-08. [Online]. Available: https://owasp.org/Top10/#welcome-to-the-owasp-top-10-2021

[5] P. Bisht, P. Madhusudan, and V. Venkatakrishnan, "Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks," ACM Transactions on Information and System Security, vol. 13, no. 2, p. 14, Mar. 2010.

[6] N. Chomsky, "On certain formal properties of grammars," Information and control, vol. 2, no. 2, pp. 137–167, Jun. 1959. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0019995859903626

[7] P. S. Landweber, "Decision problems of phrase-structure grammars," Electronic Computers, IEEE Transactions on, vol. EC-13, no. 4, pp. 354–362, Aug. 1964.

[8] J. E. Hopcroft and J. D. Ullman, Formal Languages and Their Relation to Automata. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1969.

[9] S. Ginsburg and E. H. Spanier, "Quotients of context-free languages," Journal of the ACM, vol. 10, no. 4, pp. 487–492, Oct. 1963.

[10] S. Ginsburg and S. A. Greibach, "Deterministic context free languages," Information and Control, vol. 9, no. 6, pp. 620–648, Dec. 1966. [Online]. Available: https://doi.org/10.1016/S0019-9958(66)80019-0

[11] G. Rozenberg, "Arto salomaa, editors. 1997. handbook of formal languages, volume 1 word, language, grammar."

[12] P. R. Asveld and A. Nijholt, "The inclusion problem for some subclasses of context-free languages," Theoretical computer science, vol. 230, no. 1-2, pp. 247–256, 2000.

[13] D. J. Rosenkrantz and R. E. Stearns, "Properties of deterministic top-down grammars," Information and Control, vol. 17, no. 3, pp. 226–256, 1970.

[14] M. M. Geller and M. A. Harrison, "On lr(k) grammars and languages," Theoretical Computer Science, vol. 4, no. 3, pp. 245–276, 1977.

[15] G. Sénizergues, "The equivalence problem for deterministic pushdown automata is decidable," in International Colloquium on Automata, Languages, and Programming, P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, Eds., Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 671–681.

[16] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, "Security applications of formal language theory," IEEE Systems Journal, vol. 7, no. 3, pp. 489–500, 2013.

[17] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," SIGPLAN Not., vol. 41, no. 1, pp. 372–382, Jan. 2006. [Online]. Available: https://doi.org/10.1145/1111320.1111070

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

[19] R. J. Hansen and M. L. Patterson, "Guns and butter: Towards formal axioms of input validation," 2005.

[20] F. Momot, S. Bratus, S. M. Hallberg, and 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.

[21] D. Kurilova, A. Potanin, and J. Aldrich, "Wyvern: Impacting software security via programming language design," in Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools, 2014, pp. 57–58.

[22] T. Dullien, "Weird machines, exploitability, and provable unexploitability," IEEE Transactions on Emerging Topics in Computing, vol. 8, no. 2, pp. 391–403, 2017.

[23] B. Beizer, Software testing techniques. Dreamtech Press, 2003.

[24] G. Hoglund and G. McGraw, Exploiting software: How to break code. Pearson Education India, 2004.

[25] R. W. Sebesta and N. D. Jones, "Parsers for indexed grammars," International Journal of Computer & Information Sciences, vol. 7, no. 4, pp. 345–359, Dec. 1978. [Online]. Available: https://doi.org/10.1007/BF00991819

[26] S. Lucks, N. M. Grosch, and 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.

[27] J. E. Hopcroft, "On the equivalence and containment problems for context-free languages," Mathematical systems theory, vol. 3, no. 2, pp. 119–124, 1969.

[28] M. Latteux and P. Turakainen, "On characterizations of recursively enumerable languages," Acta Informatica, vol. 28, no. 2, pp. 179–186, 1990. [Online]. Available: http://dx.doi.org/10.1007/BF01237236

[29] D. E. Knuth, "On the translation of languages from left to right," Information and Control, vol. 8, no. 6, pp. 607–639, Dec. 1965. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0019995865904262

APPENDIX A
DEFINITIONS FOR MULTIPLE BLANKS

Let us expand the previous definitions (IV.1, IV.2, and IV.4) to templates with m blanks.

Definition A.1 (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 injections of L associated to a template $\mathbf{t}$ as the language defined by:

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

Definition A.2 (Expected injections of G for intents $\boldsymbol{\iota}$)**.** Let G be a formal grammar, $m \geq 1$ and $\boldsymbol{\iota} = (\iota_1, \iota_2, \ldots, \iota_m) \in$

$(\Delta^+)^m$. The language of expected injections of G for intents $\boldsymbol{\iota}$, noted $E(G, \boldsymbol{\iota})$, is defined as:

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

Definition A.3 (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 intent-equivalent to $\boldsymbol{\iota}$ if:

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

In the following, we succinctly extend the definitions of intent-security from Section V to injections in multiple-blanks templates.

Definition A.4 (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:

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

Definition A.5 (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 A.6 (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 $\boldsymbol{\iota} \in (\bigcup_{k=1}^{n} \Delta^k)^m$, $\delta I(G, \boldsymbol{\iota}) = \varnothing$.

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

Definition A.7 (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.

Definition A.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.:

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

APPENDIX B
PROOFS

Proof of IV.14: Let $G_1 = (N_1, T, R_1, S_1)$ and $G_2 = (N_2, T, R_2, S_2)$ be two context-free 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:

$$\begin{array}{lll} S_3 \to \#_1 S_1 & \qquad & S_3 \to \#_1 S_2' \\ S_2' \to S_2 & & S_2' \to \#_2 \end{array}$$

Due to the rules in R', $F(L(G_3), (\#_1, \epsilon)) = L(G_1) \cup L(G_2) \cup \{\#_2\}$. Let us 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:

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

The grammar G_3 is context-free, so if the intent-equivalence for context-free grammars were decidable, then $L(G_1) \subseteq L(G_2)$ would be decidable. Because the latter is undecidable [27], we conclude that the intent-equivalence is undecidable for context-free grammars. ∎

Proof of Lemma IV.10: This proof relies on the definition of LR(k) grammars proposed by [14] (Def 2.1) that 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 $A = A', \beta = \beta', |\alpha\beta| = |\alpha'\beta'|$.

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:

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

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 $LR(k)$. ∎

Proof of V.8: 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 $\iota \in \Delta^+$. ∎

Example B.1. *Let two context-free grammars $G_1 = (N, T, R_1, S)$ and $G_2 = (N, T, R_2, S)$. 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 $\delta I(G_2, M) = \varnothing$. Finally, $\delta I(G_1, M) = \{bb\}$, $\delta I(G_2, M) = \varnothing$, so $\delta I(G_1, M) \not\subseteq \delta I(G_2, M)$.

Proof of V.9:

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

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

Proof of Proposition V.13:

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 $S_2 \to \hat{v}S_2 v$ to obtain $\gamma_1 = \hat{v}avdv$. The sequence γ_1 ends with a palindrome vdv that ends with v, hence the injection is $\hat{v}a$ 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 $S_2 \to \hat{a}S_2 a$ 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:

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

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 [8]. ∎

Lemma B.2. *For every recursively enumerable language L, there exist a regular set $\mathcal{R}$ and two morphisms g and h, such that*

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

Proof of Proposition B.2: This proof is based on the proof of a similar theorem in [28]. 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 VI.

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

TABLE VI: 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 us 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, $S = \alpha_a$, $\beta_a = u_1\alpha_b v_1$ and $u_1\beta_b v_1 = u_2\alpha_c v_2$. In the end, $h_G(w)\backslash g_G(w)$ is the last sentential form of the derivation, in this case $u_2\beta_c v_2$. The quotient ensure that the derivation is valid. The theorem of [28] states that this quotient, when intersected with T^*, recreates exactly the words of the initial language and only those, i.e.,

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

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 us show that the previous theorem holds if we replace $\mathcal{R}^1_G = A(B\Delta^*R\Delta^*)^*\overline{B}$ by $\mathcal{R}^2_G = A(B\Delta^*R\Delta^*)^+\overline{B}$:

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

To ensure that all derivations will end with words, let us 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 $\mathcal{R}^3_G = A(B\Delta^*R\Delta^*)^*(BT^*R_fT^*)\overline{B}$. Let us show that the intersection with T^* is useless with this regular set by showing that $\forall w \in \mathcal{R}^3_G, h_G(w)\backslash g_G(w) \in T^*$.

Let $w \in \mathcal{R}^3_G$, $u, v \in T^*$ 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 $h_G(w)\backslash g_G(w) = u\beta v \in T^*$, we can conclude that $\{h_G(w)\backslash g_G(w) \mid w \in \mathcal{R}^3_G\} \subseteq T^*$.

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

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

What is left to prove is that $L \subseteq \{h_G(w)\backslash g_G(w) \mid w \in \mathcal{R}^3_G\}$. Let w_G be a word of L. There exists $w \in \mathcal{R}^2_G$ such that $w_G = h_G(w)\backslash g_G(w)$. Let us 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 $w_G = h_G(w)\backslash g_G(w) = u\beta v$. 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 $w_G \in \{h_G(w)\backslash g_G(w) \mid w \in \mathcal{R}^3_G\}$.

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

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:

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

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

Proof of V.14: 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 $L = \{g(w)/h(w) \mid w \in \mathcal{R}\}$ as defined in the proof of Lemma B.2. Our goal is to create a

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

deterministic context-free grammar G and a symbol $\#$ such that its unexpected injection language $\delta\mathcal{I}(G, \#)$ is in the form $g(w)/h(w)$. 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 us 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 us 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 VI). 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 4: the words of L_3 are accepted by empty stack.

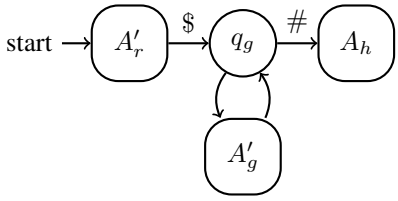


Fig. 4: A summarized DPDA for the language L_3

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

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

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 $\{w^r\$\backslash m / h(w) \mid w \in \mathcal{R}, m \in L_1\} = \{\#\}$. Passing from line (6) to line (7) is justified by the Lemma B.2.

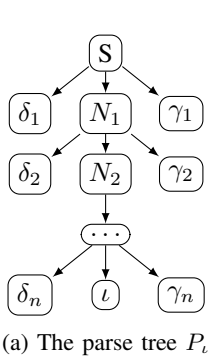
Finally, we constructed the deterministic context-free language L_3 with a deterministic grammar G such that $\delta I(G, \#) = L$. So $L \in \delta\mathcal{I}^1(G)$.

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 $n \geq 1$.

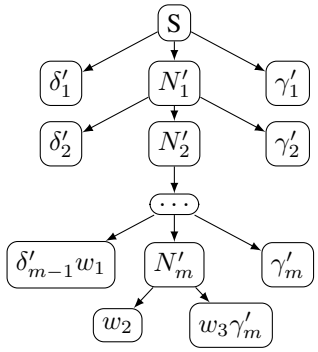
This is a direct consequence of a result shown by [29]: LR(0) languages are exactly the languages recognized by a deterministic pushdown automaton that accepts by empty stack. This is the case of the automaton constructed in the previous proof, so L_3 is an LR(0) language. In fact, we can modify the 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 $L = \delta I(G, \iota)$. ■

Proof of VI.1:

Let $p\iota s$ be a sentential form of G such that $p, s \in T^*$ and $\iota \in \Delta$. Let P_ι be the parse tree of $p\iota s$ and $S, N_1, N_2, \dots N_n$ the sequence of nodes in the branch that ends with ι (for the sake of brevity, N_i will also denote the symbol that labels the node N_i). For each node N_i ($1 \leq i \leq n$), denote δ_i the symbols on the left of the symbol N_i and γ_i the symbols of the right of the symbol N_i. By construction, $N_n = \iota$. This parse tree is represented in Figure 5a. Some of these sentential forms may be empty. We can remark that $\delta_1\delta_2 \dots \delta_n \Rightarrow^* p$ in this parse tree. Conversely, $\gamma_1\gamma_2 \dots \gamma_n \Rightarrow^* s$.



(a) The parse tree P_ι



(b) The parse tree P_w

Fig. 5: Parse trees used in the proof of VI.1

Let $w \in T^*$ such that pws is a word of $L(G)$. Let P_w be the parse tree of pws. Our goal is to show that these two parse trees are in fact similar; more precisely, that at each stage k: $\delta_k = \delta'_k$, $\gamma_k = \gamma'_k$, $N_k = N'_k$ and finally that $\iota \Rightarrow^* w$. This

proof shows that the first stage of these two trees are identical; similar reasoning can be applied recursively.

Let r_ι be the rule used at the root of P_ι and r_w be the rule used at the root of P_w. First show that $r_w = r_\iota$.

First case: $\delta_1 \neq \epsilon$. Since G is epsilon-free, there exists $w' \in T^+$ such that $\delta_1 \Rightarrow^* w'$. Since the grammar is $LL(1)$, there is only one rule that can be used from the root node and that begins with the first symbol of w'. This rule doesn't depend on ι nor w, so we can conclude that $r_\iota = r_w$. The case $\gamma_1 \neq \epsilon$ is similar and relies on the fact that G is $RR(1)$.

Second case: $\delta_1 = \epsilon$ **and** $\gamma_1 = \epsilon$. So it means that there exists a rule in the form $S \to \iota$. Due to the hypothesis on G, this is the only rule that can be applied to S, so $r_\iota = r_w$.

By definition of δ_1, N_1 and γ_1, $r_\iota = S \to \delta_1 N_1 \gamma_1$. Let $\delta'_1, \phi_1, \gamma'_1 \in \Delta^*, p', s' \in T^*$ such that $\delta'_1 \phi_1 \gamma'_1 = \delta_1 N_1 \gamma_1$, $\phi_1 \Rightarrow^* p'ws'$ in P_w and the length of ϕ_1 is minimal. Our goal is to prove that $\delta_1 = \delta'_1$, $\gamma_1 = \gamma'_1$ and $\phi_1 = N_1$.

Let us first prove that δ_1 is a prefix of δ'_1 and that γ_1 is a suffix of γ'_1. Let A be the first symbol of δ_1, $w_A \in T^*$ the derivation of A in P_ι and $w'_A \in T^*$ the derivation of A in P_ι. Since A is the first symbol of δ_1, w_A is a prefix of p, i.e. $w_A p'' = p$. For the same reason, w'_A is a prefix of pws. We can conclude that either w_A is a strict prefix of w'_A, w_A is a strict prefix of w'_A or $w_A = w'_A$. Since $L(G_A)$ is prefix-free, the two first possibilities are excluded so $w_A = w'_A$. Therefore $w'_A p'' = p$.

Let us prove that A is not the first symbol of ϕ_1 by contradiction. Assume A is the first symbol of ϕ_1, i.e. $\phi_1 = A\phi'_1$. Then $\phi_1 \Rightarrow^* w_A p'' ws$ and $\phi'_1 \Rightarrow^* p'' ws$. It is in contradiction with the assumption that the length of ϕ_1 is minimal: so A is not the first symbol of ϕ_1. Therefore, A is the first symbol of δ'_1.

This reasoning can be applied once more to show that the second symbol of δ_1 is also the second symbol of δ'_1, etc. Finally, it shows that δ_1 is a prefix of δ'_1. A similar reasoning shows that γ_1 is a suffix of γ'_1. So there exist δ''_1 and γ''_1 such that $\delta_1 \delta''_1 = \delta'_1$ and $\gamma''_1 \gamma_1 = \gamma'_1$. Since $\delta_1 N \gamma_1 = \delta'_1 \phi_1 \gamma'_1 = \delta_1 \delta''_1 \phi_1 \gamma''_1 \gamma_1$, we can conclude that $N = \delta''_1 \phi_1 \gamma''_1$.

Let us now prove $N_1 = \phi_1$ by contradiction, by assuming $\phi_1 = \epsilon$. In that case, it follows that either $\delta''_1 = N_1$ or $\gamma''_1 = N_1$. Assume that $\delta''_1 = N_1$ (the case $\gamma''_1 = N_1$ is similar). Denote p', s' such that $N_1 \Rightarrow^* p' \iota s'$; remark that on the other hand $N_1 = \delta''_1 \Rightarrow^* p'''$ for some p'''. Let $u \in T^*$ such that $\iota \Rightarrow^* u : |u| \geq 1$ because G is epsilon-free. So p''' is a strict prefix of $p'us'$. It is in contradiction with the assumption that $L(G_{N_1})$ is prefix-free. So $\phi_1 = N_1$.

Finally, we proved that $\delta_1 = \delta'_1$, $\gamma_1 = \gamma'_1$, $N_1 = N'_1$. The same reasoning can be applied to the other levels of the parse trees as well. We obtain finally that $\phi_n = \iota$ and $\phi_n \Rightarrow^* w$, so $\iota \Rightarrow^* w$. We can conclude that G is intent-secure. ■

Proof of VI.5: Consider a grammar G that has been modified by adding new opening and closing terminals, such that each rule $r = A \to \alpha$ is augmented with a unique opening terminal o_r and a closing terminal c, resulting in $A \to o_r \alpha c$. Note that the closing terminal is not necessarily unique: the same closing terminal may be used in multiple rules.

Consider the derivations $S \Rightarrow^* \alpha\iota\beta$ and $S \Rightarrow \alpha w \beta$ with $\iota \in \Delta$. We must prove that $\iota \Rightarrow^* w$.

We proceed by applying two induction principles: one on the derivation $S \Rightarrow^* \alpha\iota\beta$, and the other on $S \Rightarrow \alpha w\beta$. In the following case analysis, we follow these derivations in a synchronized fashion.

Base case for $S \Rightarrow^* \alpha\iota\beta$: if $S = \alpha\iota\beta$, then $\alpha = \beta = \epsilon$, so $S = \iota$ and we can conclude directly.

Inductive step for $S \Rightarrow^* \alpha\iota\beta$: Suppose $S \to \delta_1 \cdots \delta_n$ with $\delta_i \in V$ and $\delta_i \Rightarrow^* \gamma_i$ such that $\gamma_1 \cdots \gamma_n = \alpha\iota\beta$. By the induction hypothesis, we assume the statement holds for each pair (δ_i, γ_i).

Base case for $S \Rightarrow \alpha w\beta$: If $S = \alpha w\beta$, then $|\alpha w\beta| = 1$. However, the grammar is strictly monotonic ($|\mu| \geq 2$ for each $A \to \mu \in G$). Then $n \geq 2$ and we have a contradiction.

Inductive step for $S \Rightarrow \alpha w\beta$: Suppose $S \to \delta'_1 \cdots \delta'_m$ with $\delta'_i \in V$ and $\delta'_i \Rightarrow^* \gamma'_i$ such that $\gamma'_1 \cdots \gamma'_n = \alpha w\beta$.

Since the intent cannot be a marker, we have $\delta_1 = \delta'_1$. Therefore, the first rule of the derivation $S \Rightarrow^* \alpha a\beta$ is the same as the first rule of the derivation $S \Rightarrow \alpha w\beta$. Hence, $n = m$ and $\delta_i = \delta'_i$ for all i.

One of the elements δ_i derives the symbol ι. Let us denote this symbol by δ_k. Note that δ_k derives ι, but it may also derive part of α and β. We have: $\alpha = \alpha_1\alpha_2$, $\beta = \beta_1\beta_2$, $\delta_1 \cdots \delta_{k-1} \Rightarrow^* \alpha_1$, $\delta_k \Rightarrow^* \alpha_2\iota\beta_1$ and $\delta_{k+1} \cdots \delta_n \Rightarrow^* \beta_2$.

By rewriting, we obtain $S \Rightarrow \alpha_1\alpha_2 w \beta_1\beta_2$. Given that $\delta_1 \cdots \delta_{k-1} \Rightarrow^* \alpha_1$ and $\delta_{k+1} \cdots \delta_n \Rightarrow^* \beta_2$ it follows that $\delta_k \Rightarrow^* \alpha_2 w \beta_1$.

Using the induction hypothesis on $\delta_k \Rightarrow^* \alpha_2\iota\beta_1$ and $\delta_k \Rightarrow^* \alpha_2 w \beta_1$, we can conclude directly. ■