JunId: Fast Intersection with Incomplete Sentences

Computing injection grammars from templates

Eric Alata Pierre-François Gimenez

LangSec 2026

Opening Example

Complete the following sentence

Paris is to □ what London is to □

Opening Example

Complete the following sentence

Paris is to □ what London is to □

A natural answer is

Paris is to France what London is to England

Opening Example

Complete the following sentence

Paris is to □ what London is to □

A natural answer is

Paris is to France what London is to England

Another answer is also a valid sentence

Paris is too crowded for you, and that's what London is to me

Injection Attacks

Idea

An injection attack uses an input to change the semantics of a sentence

This is not specific to SQL

- Interpreted languages: Bash, JavaScript, Python
- Formats: XML, LDAP filters, HTML, CSS
- Protocols: SMTP, LDAP

Classical Software Example

A developer writes a query template

```
SELECT id FROM user WHERE login='□' AND password='□'
```

Classical Software Example

A developer writes a query template

```
SELECT id FROM user WHERE login='□' AND password='□'
```

A malicious input may close a string and add syntax

```
login = admin     password = ' OR 1=1-
```

Classical Software Example

A developer writes a query template

```
SELECT id FROM user WHERE login='□' AND password='□'
```

A malicious input may close a string and add syntax

```
login = admin      password = ' OR 1=1-
```

The question is not only: "Can we find this attack?"

The stronger question is: "Can we compute all valid completions?"

Goal

We want a static analysis component

- Input: a grammar for a language
- Input: a template, with fixed parts and blanks
- Output: a grammar for the strings that can fill the blanks

This grammar can support

- vulnerability detection, by checking whether an attack token is supported
- proof of absence, by proving that no dangerous command is supported
- fuzzing, by using the grammar as a supported payload generator

Running Example : SMTP Grammar

We use a small grammar for SMTP-like sessions

$$
\begin{aligned}
\textit{Session} &\rightarrow \textit{AuthSession} \mid \textit{MailSession} \\
\textit{AuthSession} &\rightarrow \textit{Ehlo}\ \textit{AuthCmd} \\
\textit{MailSession} &\rightarrow \textit{Ehlo}\ \textit{MailCmd}\ \textit{RcptCmd}\ \textit{DataCmd} \\
\textit{Ehlo} &\rightarrow \texttt{EHLO SP}\ \textit{Domain}\ \texttt{CRLF} \\
\textit{Domain} &\rightarrow \texttt{example.org} \\
\textit{AuthCmd} &\rightarrow \texttt{AUTH SP}\ \textit{Mechanism}\ \texttt{CRLF} \\
\textit{MailCmd} &\rightarrow \texttt{MAIL FROM:<}\ \texttt{ADDRESS}\ \texttt{>}\ \texttt{CRLF} \\
\textit{RcptCmd} &\rightarrow \texttt{RCPT TO:<}\ \texttt{ADDRESS}\ \texttt{>}\ \texttt{CRLF} \\
\textit{DataCmd} &\rightarrow \texttt{DATA CRLF}
\end{aligned}
$$

Blue bold symbols are terminals Other symbols are nonterminals

Running Example : Template Intent

A template is a sentence with blanks

```
EHLO example.org CRLF MAIL FROM:<□> CRLF
RCPT TO:<ADDRESS> CRLF DATA CRLF
```

The developer usually expects the blank to be one email address

- Legitimate input: `ADDRESS`
- Malicious input: `ADDRESS> CRLF RCPT TO:<ADDRESS`

The malicious input adds a new recipient command

```
EHLO example.org CRLF MAIL FROM:<
ADDRESS> CRLF RCPT TO:<ADDRESS
> CRLF RCPT TO:<ADDRESS> CRLF DATA CRLF
```

The Core Language Problem

Let G be a context-free grammar

Let an incomplete sentence be a sequence of fixed parts separated by blanks

With one fixed part r, the problem is

$$L(G) \cap \Sigma^* r \Sigma^*$$

This is a basic block for template analysis

- Run it for fixed parts of the template
- Mark symbols before, inside, and after each fixed part
- Then project the marked grammar to obtain possible injections

Why Existing Algorithms Are Too Slow

The literature usually treats a more general problem

It studies decidability of

$$\text{context-free language} \cap \text{regular language}$$

The regular model is often given as an automaton

The standard construction intersects a CFG with this automaton

It builds nonterminals of the form

$$(q_a, A, q_b)$$

This means

$$A \text{ derives a word read by paths from } q_a \text{ to } q_b$$

The construction is correct, but the complexity is high and many symbols are useless

In the paper, this makes the baseline time out on JSON and SQLite grammars

Contributions

The paper makes two contributions

First contribution: Faster general CFG/NFA intersection

- Keep the classical cross-product idea
- Reject useless triplets with sound predicates
- Implemented predicates: FirstP, FirstLastP, FirstLastReachP

Second contribution: JunId for incomplete sentences

- Do not build a full automaton product
- Identify where a fixed radical crosses grammar rules
- Add no unnecessary nonterminals

JunId is the focus of the next slides

What JunId Builds

Input

$$G = (N, \Sigma, R, S) \qquad r \in \Sigma^*$$

Output

$$G_r \quad \text{such that} \quad L(G_r) = L(G) \cap \Sigma^* r \Sigma^*$$

The output grammar keeps only words of G that contain r

For templates, this grammar is later used with markings to isolate the blank parts

Three Kinds of New Nonterminals

For each original nonterminal A

- $A_r^{\diamond}$ derives words from A that contain the full radical r
- $A_p^{\triangleleft}$ derives words from A that end with a non-empty prefix p of r
- $A_s^{\triangleright}$ derives words from A that start with a non-empty suffix s of r

These symbols are created only when they are useful

Precomputation

JunId first transforms the grammar to 2NF

Then it computes three predicates for every relevant part of r

$$\begin{array}{rcl} \mathrm{Rad}(A, w) & : & A \text{ can derive a word containing } w \\ \mathrm{Pre}(A, w) & : & A \text{ can derive a word starting with } w \\ \mathrm{Suf}(A, w) & : & A \text{ can derive a word ending with } w \end{array}$$

The paper computes these predicates with a modified CYK parser

The implementation uses the cubic version The complexity statement uses Valiant's parser

Running Example : First Radical

We start with the first fixed part of the template

$$r = \texttt{EHLO example.org CRLF MAIL FROM:<}$$

Q1. Does the axiom *Session* derive a word containing r? Yes

$\Rightarrow$ We create a new covering symbol: $\mathit{Session}_r^{\diamond}$

Running Example : Axiom Rules

Now we inspect the rules of *Session*, one by one

The axiom has two rules $Session \to AuthSession$ $\quad$ $Session \to MailSession$

Q2. Does the right-hand side of $Session \to AuthSession$ derive a word containing r? No

$\Rightarrow$ Stop for this rule

Q3. Does the right-hand side of $Session \to MailSession$ derive a word containing r? Yes

$\Rightarrow$ The right-hand side has only one symbol, so it has the responsibility for r:

$$Session_r^\diamond \to MailSession_r^\diamond$$

Running Example : Mail Session Split

Now we inspect the rules of *MailSession*, one by one

MailSession has one rule $\mathit{MailSession} \to \mathit{Ehlo}\ \mathit{MailCmd}\ \mathit{RcptCmd}\ \mathit{DataCmd}$

Q4. Does the right-hand side derive a word containing r? Yes

$\Rightarrow$ There is only one useful split:

$$r = \underbrace{\texttt{EHLO example.org CRLF}}_{r_1} \mid \underbrace{\texttt{MAIL FROM:<}}_{r_2} \mid \quad \mid \quad \mid$$

$$\mathit{Ehlo} \qquad \mathit{MailCmd} \qquad \mathit{RcptCmd} \quad \mathit{DataCmd}$$

MailCmd is wider because it generates `MAIL FROM:<` and then more syntax

Running Example : First Junction

JunId creates a junction rule

$$MailSession_r^{\diamond} \rightarrow \overline{\texttt{EHLO example.org CRLF}}^{\,r}\, MailCmd_{r_2}^{\triangleright}\, \overline{RcptCmd}^{\,s}\, \overline{DataCmd}^{\,s}$$

This rule says exactly where the radical is

- Here, *Ehlo* must generate `EHLO example.org CRLF`, so we substitute it
- The start of *MailCmd* contributes `MAIL FROM:<`
- The remaining commands are outside the first radical

The next question is How can *MailCmd* start with `MAIL FROM:<`?

Running Example : Mail Command Suffix

MailCmd has one rule *MailCmd* $\rightarrow$ `MAIL FROM:< ADDRESS > CRLF`

The rule can start with `MAIL FROM:<`

JunId creates the closing rule

$$MailCmd^{\triangleright}_{r_2} \rightarrow \overline{\texttt{MAIL FROM:<}}^{\,r}\ \overline{\texttt{ADDRESS}}^{\,s}\ \overline{\texttt{>}}^{\,s}\ \overline{\texttt{CRLF}}^{\,s}$$

$MailCmd^{\triangleright}_{r_2}$ means the rule must start with r_2, then the rest is after the radical

Running Example : Generated Rules

For $r = \texttt{EHLO example.org CRLF MAIL FROM:<}$

$$
\begin{aligned}
\mathit{Session}_r^{\diamond} &\rightarrow \mathit{MailSession}_r^{\diamond} \\
\mathit{MailSession}_r^{\diamond} &\rightarrow \mathit{Ehlo}_{r_1}^{\triangleleft}\, \mathit{MailCmd}_{r_2}^{\triangleright}\, \overline{\mathit{RcptCmd}}^{\,s}\, \overline{\mathit{DataCmd}}^{\,s} \\
\mathit{MailCmd}_{r_2}^{\triangleright} &\rightarrow \overline{\texttt{MAIL FROM:<}}^{\,r}\, \overline{\texttt{ADDRESS}}^{\,s}\, \overline{\texttt{>}}^{\,s}\, \overline{\texttt{CRLF}}^{\,s}
\end{aligned}
$$

Suffix copies of initial rules are also added

$$
\begin{aligned}
\overline{\mathit{RcptCmd}}^{\,s} &\rightarrow \overline{\texttt{RCPT TO:<}}^{\,s}\, \overline{\texttt{ADDRESS}}^{\,s}\, \overline{\texttt{>}}^{\,s}\, \overline{\texttt{CRLF}}^{\,s} \\
\overline{\mathit{DataCmd}}^{\,s} &\rightarrow \overline{\texttt{DATA}}^{\,s}\, \overline{\texttt{CRLF}}^{\,s}
\end{aligned}
$$

The original grammar is not copied Only useful decorated rules are added

Running Example : From Fixed Parts to Blanks

For the template

$$\texttt{EHLO example.org CRLF MAIL FROM:<}\square\texttt{> CRLF}$$
$$\texttt{RCPT TO:<ADDRESS> CRLF DATA CRLF}$$

the first fixed part is

$$r_1 = \texttt{EHLO example.org CRLF MAIL FROM:<}$$

A later fixed part is

$$r_2 = \overline{\texttt{>}}^s\overline{\texttt{CRLF}}^s\overline{\texttt{RCPT TO:<}}^s\overline{\texttt{ADDRESS}}^s\overline{\texttt{>}}^s$$

JunId is run on these fixed parts Markings then isolate the grammar of the blank

- The second radical is rewritten with $\bar{\cdot}^s$ symbols because it must occur in the suffix of the previous radical

Running Example : Security Query

Once the injection grammar is computed, we can ask security questions

For example

Can the blank contain `CRLF RCPT TO:`?

If yes, the input can add a new recipient command

This is stronger than testing one payload

It characterizes all syntactically possible completions for the blank

Running Example : Injection Language

For the SMTP template, a safe value should stay inside one address

But the grammar may also allow a completion that opens a new command

```
ADDRESS> CRLF RCPT TO:<ADDRESS
```

This changes the structure of the SMTP session

The output grammar is useful because it is not a single payload

It describes all payloads compatible with the surrounding syntax

Step 1: Follow the Full Radical

Covering rule

Start from $S_r^\diamond$

If $A \to \alpha B \beta$ and B can contain r, create

$$A_r^\diamond \to \overline{\alpha}^{\,p} B_r^\diamond \overline{\beta}^{\,s}$$

B is responsible for the radical The symbols α are before it The symbols β are after it

This is a breadth-first search over useful covering nonterminals

Step 2: Identify Junction Rules

Junction rule

Sometimes no child contains the whole radical

The radical may cross a boundary in the right-hand side of a rule

JunId creates opening and closing nonterminals for such cases

We now specialize this idea to a binary rule in 2NF

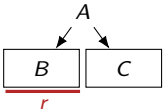
Step 2: Identify Junction Rules – Four Cases

This is a key result for the correctness of the algorithm

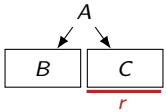
For rules in 2NF, suppose

$$A \to B\ C \qquad \text{and} \qquad r = r_1 r_2$$

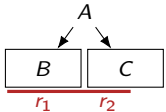
These five cases are exhaustive for binary 2NF rules



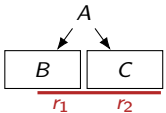
$A^{\diamond}_{r} \to \overline{r}^{\,r}\,\overline{C}^{\,s}$



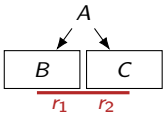
$A^{\diamond}_{r} \to \overline{B}^{\,p}\,\overline{r}^{\,r}$



$A^{\diamond}_{r} \to \overline{r_1}^{\,r}\,C^{\triangleright}_{r_2}$



$A^{\diamond}_{r} \to B^{\triangleleft}_{r_1}\,\overline{r_2}^{\,r}$



$A^{\diamond}_{r} \to B^{\triangleleft}_{r_1}\,C^{\triangleright}_{r_2}$

For rules in 2NF, this slide covers all cases, and the article states a more general theorem

Step 3: Finish Prefixes and Suffixes

Opening and closing rules

After a junction, JunId must complete the missing side of the radical

For a closing nonterminal $A_s^{\triangleright}$

- keep rules where A can start with s
- recursively shorten s
- stop when the whole suffix of r is generated

Opening nonterminals $A_p^{\triangleleft}$ are processed symmetrically

Correctness

The constructed grammar is exact

$$L(G_r) = L(G) \cap \Sigma^* r \Sigma^*$$

The proof gives two security properties

- If an injection exists, it can be generated by the grammar built by JunId
- If this grammar generates a string, then this string is a valid injection

The corresponding proof is verified with Coq (this matters for security)

Proof details and Coq code are available in the article

No Unnecessary Nonterminals

Proposition

If the input grammar has no unnecessary nonterminals, then JunId adds no unnecessary nonterminals

This is the central practical property

- $A_r^{\diamond}$ is created only if A can contain r
- $A_p^{\triangleleft}$ is created only if A can end with p
- $A_s^{\triangleright}$ is created only if A can start with s

Complexity

With Valiant's parser, the paper states

$$O(|G| \times |r|^{\omega}) \qquad 2 \leq \omega < 2.373$$

The expensive step is the parsing precomputation

Under the clique-algorithm hypothesis used in the paper, this is asymptotically optimal

The OCaml implementation uses CYK, so its implemented bound is cubic in $|r|$

Experiment: Incomplete Sentences

Mean duration of successful intersections

Algorithm	anbns	small	JSON	SQLite
Baseline	4.48s	13.23s	17.05s	timeout
FirstLastReachP	0.11s	5.10s	0.91s	127.45s
Lang88 top-down	**0.006s**	64.33s	0.13s	1728.64s
JunId	0.01s	**0.21s**	**0.04s**	**2.1s**

JunId had no timeout in this experiment

Experiment: Realistic Scenario

A PHP program builds an SQLite query from two inputs

PHP template

```
<?php $iban = $_POST["iban"]; $date = $_POST["date"]
if(strlen($iban)==14 and
    preg_match("/^[0-9-]*$/",$date)) {
  $result=sqlite_query(
"SELECT * FROM USERS WHERE IBAN=
'$iban' and date='$date'"); } [...] ?>
```

- JunId is run three times for the fixed parts of the template
- FirstLastReachP then intersects with the input filters `/^[0-9-]*$/ & .{14}`
- Total time: 11.5 seconds
- Result: 775 nonterminals, 138 terminals, 1130 rules

Lang88 top-down produced a much larger grammar and then crashed during the second step

Takeaways

- Template security can be phrased as a language-intersection problem
- Standard CFG/NFA intersection is correct but often too large
- JunId uses the structure of incomplete sentences
- Its key idea is to identify where the fixed radical crosses grammar rules
- It builds an exact grammar without unnecessary nonterminals
- Usage: JunId generates the injection grammar from a template and a grammar
- Usage: FirstLastReachP checks that developer filters keep only intended values

Limitations and Next Steps

- The implementation uses a custom grammar format
- It does not separate lexer and parser engineering
- Multiple radicals are handled by successive calls and markings
- Integration into IDEs or security tools still requires engineering work

Main message

JunId makes exact grammar-based template analysis practical for real computer-language grammars

Thank you