2017 IEEE Symposium on Security and Privacy Workshops

Eliminating Input-Based Attacks by Deriving Automated Encoders and Decoders from Context-Free Grammars

Tobias Bieschke[1], Lars Hermerschmidt[1,2], Bernhard Rumpe[1] and Peter Stanchev[1]
Software Engineering, RWTH Aachen University, Germany[1]
AXA Konzern AG, Germany[2]

***Abstract*—Software systems nowadays communicate via a number of complex languages. This is often the cause of security vulnerabilities like arbitrary code execution, or injections. Whereby injections such as cross-site scripting are widely known from textual languages such as HTML and JSON that constantly gain more popularity. These systems use parsers to read input and unparsers write output, where these security vulnerabilities arise. Therefore correct parsing and unparsing of messages is of the utmost importance when developing secure and reliable systems. Part of the challenge developers face is to correctly encode data during unparsing and decode it during parsing.**

This paper presents McHammerCoder, an (un)parser and encoding generator supporting textual and binary languages. Those (un)parsers automatically apply the generated encoding, that is derived from the language's grammar. Therefore manually defining and applying encoding is not required to effectively prevent injections when using McHammerCoder. By specifying the communication language within a grammar, McHammerCoder provides developers with correct input and output handling code for their custom language.

I. Introduction

Nowadays virtually every application runs in distributed fashion, regularly across trust boundaries. In order to communicate, components of the application need to agree on some language for messages they exchange. This language is also called protocol, or format. Countless vulnerabilities like buffer overflows, SQL injection (SQLi), and cross-site scripting (XSS) caused by "malicious" input show that handling input and output data is an underestimated challenge and major security concern in distributed applications. These weaknesses have been studied for numerous languages. However, this search is an endless journey, since all programs which implement a parser or unparser are at risk.

The parser, being the part of a program which reads input, is responsible for validating all preconditions the program expects the input to meet, i.e. to belong to the communication language and to create well-typed objects that represent the input within the program. While writing output, it is the unparser's job to ensure the output follows the structure defined by program code no matter what data the program processes. If the data is "malicious", unparsing it might result in a message that is interpreted differently by a parser then it was intended when it was unparsed.

When parsing is performed correctly, "malicious" input is rejected by the parser as it is not expected by the program. While correct unparsing assures that no input may change the output of a program into a different message from the one defined by developers in source code. To enable an (un)parser to distinguish between valid and "malicious" massages, a definition of the language it accepts or respectively produces is required. This definition is a grammar, which at most might be deterministic context free [1] to allow safe recognition of input. When developers do not define the input and output language of a program explicitly, "malicious" input is just unexpected by the developer and may drive the program into unwanted states like buffer overflows or make it produce irregular and unexpected output like SQLi and XSS. Ensuring that programs create only expected output requires them to encode all unexpected data within the output language during unparsing. Although (un)parsers are ubiquitous – just like data within programs is – correctly implementing them without security vulnerabilities is a complex task. To ease this task, parsers are generated from grammars [2], [3] or parser combinators [4]–[6] are used. These solutions are very popular and well studied for textual programming languages. However, for binary protocols and file-formats there are only a few promising attempts like Nail [7] or Hammer [8].

Pretty-printers, a variant of unparsers which focus on the layout of their textual output are derived from grammars [9] and vice versa [10]. However, these approaches do not consider attacker-controlled input to be unparsed and therefore fail to prevent injection attacks. Considering these cases requires the grammar to contain context specific encoding rules which define how arbitrary data can be encoded within the language [11]. From this enriched grammar an unparser is generated, that applies (en)coding automatically during (un)parsing. However, an (un)parser generator that supports safe input and output handling from a plain context-free grammar has yet to be created. To provide developers with a comprehensive solution for secure input handling, we present MontiCore Hammer Coder[1] – short McHammerCoder – that closes this gap. This enables developers to generate a parser and unparser for any context-free grammar to safely process

[1]McHammerCoder is available on GitHub
https://github.com/McHammerCoder/McHammerCoder

DOI 10.1109/SPW.2017.32

input and output of the defined language. To achieve this, McHammerCoder detects special cases within the language definition which can be used in a way not intended by the developer. Namely those cases which require encoding to be performed when data is unparsed. To provide an automated solution, an encoding is derived from the grammar that is automatically applied during (un)parsing.

The remainder of this paper is structured as follows: First related work is reviewed. In section III, the challenges of generating a correct encoding are analysed. Section IV introduces the aforementioned McHammerCoder parser and unparser generator. Our approach to automatically derive an encoding and apply it is presented in section V. By taking the example of HTML and DNS the approach is evaluated in section VI. Finally an conclusion is drawn in section VII.

II. Related Work

The construction of parsers [2] and unparsers [12]–[14] is a long studied area in computer science. When unparsers are used to produce easy to read output, they are also called pretty-printers. Both generating pretty-printers from context-free grammars [9] and generating context-free grammars from pretty-printers [10] is used to avoid redundancy and inconsistency [15].

Language-theoretic security (LangSec) builds upon this research to reason about security vulnerabilities caused by inadequate input handling [1], [16] and output construction [11]. According to the LangSec approach correct input handling starts by defining the input and output languages of a program. This definition can either be specified in a special grammar language or written within a programming language. The former is used by parser generators like MontiCore [17] or ANTLR [3] for pure textual or Nail [7] for binary languages, while the latter requires a parser combinator to provide an internal domain specific language (DSL) [18] to describe the grammar. One such parser combinator is Hammer [8], which also implements its own arena allocator to prevent known vulnerabilities - occurring while allocating memory for the parsed data or the parsers themselves. A concept that Nail picks up as well.

Different concepts have been developed to protect against injection attacks. An early approach to solve the problem was the development of encoding tables for each language and the manual application of those tables by developers within program code. The OWASP project for example provides a table to prevent XSS [19]. Applying encoding manually has been shown to be highly error prone [20] and in the more recent years a way of automating the process has been sought. *Nail* [7], for example, offers a way of protecting against some types of injection attacks by restricting the expressiveness of the language. This is achieved by providing a custom interface between the syntax tree representation of the input and the user to prevent unwanted interactions.

There are also multiple approaches that provide security against only a particular type of injection attacks such as *Noxes* [21] and *contrast-rO0* [22]. Noxes provides a way of securing data by acting as a proxy which is able to filter out untrusted data which may contain an injection. *Contrast-rO0* tackles the task of securing against Java deserialisation [23] attacks by providing a library automatically capable of detecting serialisable objects which pose a security threat. However, both approaches only provide a solution for a particular language and type of injection attack. MontiCoder [11] provides an automated encoding to protect against injection attacks for all context-free languages. This however requires to define the encoding together with the grammar, which is still error prone, due to the complexity of the encoding the language designer has to create.

III. Formal Problem Definition

Handling Input requires to parse it – namely create a parse tree (PT) from an input message – and unparsing it, which creates the message from the PT. Following the LangSec philosophy [1], [11], [16], [24] to prevent input based attacks (at least) the following conditions need to be fulfilled:

1) Valid input and output of a program is defined by a deterministic context-free grammar[2].
2) The parser rejects all input messages which are not part of the language defined by the grammar.
3) The program uses only information from the input PT which is meant to be variable during program execution. This is an important design decision which corresponds to the intended program behaviour.
4) For all parse trees t containing any kind of data the unparser ensures a correct parsing round-trip $parse(unparse(t)) = t$

To solve the first three points, known techniques from parser construction are utilized to provide a solution for all deterministic context-free languages. The last point, however, requires some deeper analysis to provide a correct round-trip for all deterministic context-free languages.

Since encoding during unparsing handles edge cases within the language definition to ensure a correct parsing round-trip, a systematic way of deriving all edge cases is required to create a correct encoding. The root cause of such edge cases is the misinterpretation of an unparsed message by a lexer while parsing it. This is caused by crafted input which is fed into the PT before unparsing it. Such input tricks the lexer into tokenizing the message differently then intended by the developer. This results in a different PT being created. In many cases, this changes the semantics of the transmitted message.

[2]Note that there are various syntaxes to notate a grammar, for example in a dedicated grammar language, within a programming language API, or configuration file syntax

To guarantee an automated correct parsing round-trip, a correct encoding has to be generated for each context-free language. Furthermore, this generated encoding must not create new injection opportunities. In order for an encoding to be considered correct, it has to fulfil the following properties [25]:

1) Uniquely decipherable - there has to be only one possibility to decode an encoded message. Therefore attention has to be paid not only to the process of encoding creation but also to the process of applying it during decoding and encoding.
2) Instantaneous - no encoded word should appear as prefix or suffix of another encoded word. This makes it easier to decode and makes sure that (1) is not violated.
3) Secure - all keywords that can be misinterpreted by the lexer when for a given token type must be encoded. This also includes pre- and postfixes of the keywords.

IV. McHammerCoder Parser

As mentioned before our goal is a one-stop solution for developers when faced with reading or writing data to communicate securely with other programs. More precisely we aim to provide parsers and unparsers, for any protocol – textual or binary.

The solution presented here is implemented in the parser generator McHammerCoder, which is available on GitHub. Similar to Nail, the idea is to use a parser combinator, in this case Hammer – as a basis for the generated parser. The code of a parser based on such a library is very similar to the grammar definition, due to the fact that it consists of multiple predefined parsers – each corresponding to a part of the grammar. This allows for direct translation form a grammar to an actual parser.

A. Grammar Design

The MontiCore grammar language [17] has been chosen as a basis for the McHammerCoder grammar language. It allows the description of context-free grammars for textual languages, aims for ease of use for language developers, and offers concepts for deriving the abstract syntax tree (AST) of the parsed data. The grammar is built up based on rules as shown in Listing 1. Each rule is defined through a regular expression which may contain non-terminals. These rules are divided into normal rules and token rules.

Token rules describe actual sequences of characters, while normal rules combine sequences to more complex constructs. The only normal rule in this example is the start rule (line 2). In addition there are two token rules (line 3-4). The token `A` parses a variable number of `a`'s while token `B` parses a variable number of `b`'s. The normal rule `StartRule` defines that these two tokens have to occur in a particular order. The parsed input has to consist of at least one `A` followed by any number of `B` and `A` alternately.

In addition the MontiCore grammar language has features such as inheritance known from object oriented languages to support developers in creating language families that reuse common parts of existing grammars.

```
1 grammar ExampleGrammar {
2   StartRule = A (B A)*;
3   token A = ('a')*;
4   token B = ('b')*;
5 }
```

Listing 1. MontiCore grammar example

At this point it is worth noting that binary languages are a superset of textual languages, as every character is represented as a value of a particular bit-length – in other words every textual language is isomorphic to a binary language. The main difference is that binary languages use a larger set of symbols. Therefore in the following the MontiCore grammar language is extended by such symbols. Later in this section the translation from grammars to parse trees and ASTs is discussed.

We start with integer values, both unsigned or signed, which have a length of 8, 16, 32, or 64 bits specified as `uintX` – where X denotes the length in bits and the leading "u" marks it as unsigned. Consequently `intX` denotes a signed integer of length X.

Secondly bit sequences are added, where any length between 1 and 64 bits can be parsed as one symbol resulting in one long integer value together with an extra byte denoting how many bits are relevant in that value. This is basically the way Hammer treats parsed values of variable bit-lengths. These are specified as `ubitsX` – where X can be any number between 1 and 64 and the "u" marks it as unsigned. And `bitsX` – where X can be any number between 1 and 64.

The grammar language also allows to specify either a value or a range of values that are valid for that symbol by adding this value in squared brackets or two values in square brackets separated by two dots. Thus an unsigned 8 bit integer that allows values from 5 to 20 is specified as `uint8[5 .. 20]`.

In addition to the extended set of symbols, the grammar language is also extended by a new type of rules. These are called binary rules, denoted by the prefix `binary`, and are used to define sequences of binary values, which can be reinterpreted as one value or a string of characters – allowing for the definition of tokens for different character encodings. Besides the `binary` prefix, the only difference of these rules to token rules is that they can only contain non-terminal symbols referring to other binary tokens as described before.

```
1 grammar ExampleGrammar {
2   StartRule = Float;
3   binary Float = ubits1(1) ubits31 : float;
4 }
```

Listing 2. A Binary Rule in the McHammerCoder grammar

In the example in Listing 2 the binary rule `Float` (line 3) defines a binary token representing a 32 bit value. This value is divided into two parts: the first one is a one bit unsigned value restricted to have the value `1`; the second one is a 31 bit unsigned value. By postfixing "`: float`" to the rule the resulting value is reinterpreted as a Java floating point value. Due to the rule definition it is restricted to be negative. Besides the default type mapping the user may also define custom conversions here. However this is out of scope of this work.

Notably many binary formats are built up in a non linear way. Therefore, a parser cannot parse such input from the beginning all the way to end. In these scenarios offset values within the parsed message are pointing to different positions on the input, where further data can be found. One example of this are ZIP files which consist of a header at the end of the file – containing the list of files together with an offset value which denotes the location of the file inside the ZIP. This of course is a very uncommon behaviour compared to textual languages.

To define an offset in the McHammerCoder grammar, a linear function of the form $m \cdot x + b$ is used. Here m is the scale of the offset, x is the parsed value and b shifts the offset by a static value. The result is interpreted as number of bits. By choosing m the offset is calculated as a multiple of m bits depending on the parsed value of x. This offset can be applied to three different base positions. First the beginning of the parsed data, second the end of the parsed data and third the beginning of the parsed offset value.

```
1 grammar ExampleGrammar {
2   StartRule = uint8#OffsetRule ;
3   binary Rule = uint8;
4   offset OffsetRule = Rule : -4*x-4;
5 }
```

Listing 3. An Offset Rule in the McHammerCoder grammar

Listing 3 shows an example of an offset definition in a McHammerCoder grammar. In line 2, the starting rule for the parser is defined. It only contains one uint8 value marked as an offset that is defined by the rule `OffsetRule` in line 4. That offset rule starts with the keyword `offset` to define that this offset is located relatively to either the start or the end of the input. A preceding `local` in front of the `offset` keyword would indicate that the origin of the offset is the parsed `uint8` value itself. The first part after the equals sign of the offset definition – here `Rule` – describes the rule that is applied at the offset position to continue parsing. After the `:` follows the definition of the aforementioned linear function. The solution of that function is added to the base position resulting in the actual offset position. Whether the start or the end of the input is picked as base position depends on the sign of the m in the linear function. If the sign is negative it is the end of the input. In this example m is -4 in the equation, which means that the uint8 value describes the offset as a multiple of four bits in reverse direction – starting from the end of the input. The final -4 indicates, that the origin is also shifted 4 bits backwards from the end. This approach allows the use of offsets in a controlled fashion without the need for developers to write any code in a programming language to calculate the offset.

Another aspect of parsing binary data are length-fields. Length-fields are values in the input that determine how often a specific value or sequence of values occur at a later position. This is slightly harder to embed into a parser generator, as it allows even more complex language definitions than offsets.

The easiest form of these length-fields determine the amount of data that follows right after this field. In fact Hammer already implements a parser combinator that does exactly this. It applies one parser and applies another parser as often as the unsigned value, which was parsed by the first parser, tells it to. However, Hammer does not offer a real solution if the length-field and the data-field are not directly connected or if there are multiple data-fields for one length field. The main problem with this is, that the length and the data could even be in different rules. In such a situation it can be hard to avoid scenarios where the parser tries to parse data based on a length field that has not been parsed up to that point in time. Nail for example solves this through arguments passed to the parsers of each structure-definition. The McHammerCoder parser generator picks up that idea in a slightly different way. Instead of passing arguments, length- and data-fields are named and can be matched by that name. To avoid ambiguity every length-field has to have a unique name and the data-fields for each length-field are only allowed to be subrules of rules that implement this length-field. If they are subrules of different rules these rules also have to fulfil this condition. Listing 4 shows an example of an 8 bit length field describing the length of a sequence of 16 bit integers. The `LengthName` in brackets is the name that connects the length and the data field.

```
1 grammar ExampleGrammar {
2   StartRule = uint8.L{LengthName} uint16.D{
        LengthName};
3 }
```

Listing 4. A length and data field in the McHammerCoder grammar

Besides non-linear binary formats there are also formats that use different byte orders. Usually values in Java for example are in little-endian. However, there are formats and scenarios where other byte orders are used. Thus in McHammerCoder it is possible to mark any binary value as little or big-endian, by adding `.LE` or `.BE` respectively on their right, or even change the default to either of them, by setting the corresponding option. If no specific endian is defined little-endian is used.

B. Deriving Parse Tree and Abstract Syntax Tree

Every parser has two tasks. First it rejects invalid input and second it transforms the input into an abstract syntax tree (AST) with which the program behind it can work. MontiCore derives this internal AST representation directly from the grammar. However it simplifies the AST to ease its use and therefore drops all parsed data that is not considered relevant for the application using the parser to read data. For example in textual languages white spaces which are just separators that make input, respectively output, easier to read are removed. However, removing such data creates ambiguities when unparsing an AST. Therefore McHammerCoder uses a parse tree (PT) to avoid the challenge of reintroducing removed values during unparsing.

The McHammerCoder Grammar directly translates into the resulting PT. The tree consists of a root note – containing the actual parse result as well as the parse results of the applied subparsers at any offset location. Each token forms a terminal node or leaf in the parse tree. The same holds for binary tokens. Normal rules form inner nodes, which contain sequences of other inner nodes or (binary) tokens.

V. Deriving Encoding and Decoding

As previously mentioned, unparsing without correct encoding leads to an incorrect parsing round-rip and injection vulnerabilities. The cause of this is the possibly wrong interpretation of symbols or groups of symbols that are treated as keywords in some contexts of the language. Those keywords influence the way the parser reads a message and constructs the PT. To ensure the correct interpretation by the parser, encoding is used during unparsing.

To automate the generation of an encoding for a language the first step is to extract some information from the McHammerCoder grammar. First, a list of strings that need encoding has to be extracted. This list is composed of keywords and all possible keyword parts they can be divided into. It is important to encode all keyword parts, because if neighbouring PT nodes contain one part of a keyword each, the unparsed message contains a keyword originating from the data. Therefore parsing this message leads to a different PT then the one that was used for unparsing. Assuming the following three nodes form the leaf nodes of a PT before unparsing: `foo<`, `p>bar<` and `/p>fuu` and that both `<p>` and `</p>` are keywords of the language. Then after unparsing the resulting message is `foo<p>bar</p>fuu`. When parsing this message it is divided into different tokens, because both keywords are found in the input. Therefore keywords have to be divided into subparts and those parts have to be encoded as well.

A second information that has to be extracted from the grammar is the *free symbols*. Those are symbols that are defined in a token and thus are allowed within that token. For example in the token: `token Id = ('a'..'z')+;`

```
1 grammar HTMLRed {
2       token Id =  ( 'a'..'z' | 'A'..'Z' )+;
3
4       P = "<p>"  (Alternatives)* "</p>";
5
6       Alternatives  = P | B | I | Id;
7       B = "<b>" Id "</b>";
8       I = "<i>" Id "</i>";
9 }
```

Listing 5. A reduced HTML grammar

the free symbols consist of the lower case English alphabet. Free symbols are used to represent encoded data within a valid message. The set of free symbols holds all possible symbols that are available for use in the whole language. Since the free symbols are extracted directly from the grammar there is no differentiation between symbols available for one token type or the other. To gain this differentiation the set of *usable symbols* is created from the free symbols. This set is composed from the free symbols that are available for a given token type. Furthermore, any free symbol that matches a keyword or keyword part is excluded from the usable symbol list, e.g. if `a` is both a free symbol and a keyword it is excluded from the usable symbols list of all token types where `a` is a keyword.

To illustrate what exactly is extracted from a grammar, consider the reduced HTML grammar in Listing 5. The language contains six keywords which are extracted from the grammar. The free symbols are composed of the English alphabet with upper and lower case letters based on the definition of the token `Id`. In this example the usable symbols coincide with the free symbols, because there is only one token type and no single letter is a keyword. Now that the keywords and the usable symbols have been extracted, the encoding can be created.

A. Generating an Encoding

To generate an encoding, first two symbols from the usable symbols are chosen. They represent a binary number – the first is used as an *escape symbol*. Every keyword is assigned a number and this number is encoded using a binary encoding schema. The first chosen escape symbol is used to start every encoding. Therefore it is used as a marker that announces to anyone reading the encoded message that after this symbol something encoded must follow. To make sure that requirements from section III are met, every keyword is assigned an unique number. Thus, the encoded message following the escape symbol is treated as a number, representing a keyword. Since the escape symbol carries additional meaning for the encoding and decoding process every occurrence of it has to be encoded as well.

Consider the generated encoding table in Listing 6. The escape symbol is `A` and therefore every `A` is encoded into `ABABABA`. As the second usable symbol `B` has been chosen. Those

```
A = ABABABA
/i> = ABBBBBB
<b = ABBBBBA
<p> = ABBBBAB
/b> = ABBBBAA
<i = ABBBABB
p> = ABBBABA
</b = ABBBAAB
<i> = ABBBAAA
</ = ABBABBB
</b> = ABBABBA
</i> = ABBABAB
<p = ABBABAA
i> = ABBAABB
</i = ABBAABA
<b> = ABBAAAB
< = ABBAAAA
b> = ABABBBB
</p = ABABBBA
</p> = ABABBAB
/p> = ABABBAA
> = ABABABB
```

Listing 6. Encoding for the reduced HTML grammar.

symbols represent a binary zero and a binary one. In this example `ABBBBBB` is interpreted as `000000`. The leading `A` is removed since it only denotes that this is in fact an encoded message. `ABABABA` is interpreted as `010101` or `21` in decimal.

It is important to note that a specific encoding has to be created per token type, because it is seldom the case that one encoding is able to fit in all possible contexts of a given language without causing vulnerabilities. Thus multiple encodings are generated – one encoding per token type. To ensure that the encoding does not change the token type when applying it, the type of each generated encoding is checked to verify that it still corresponds to the type of the token that it is meant for. If it changes the token type, the encoding is discarded and another pair of escape symbol and usable symbol is chosen. This is repeated until there are no more possible pairs of usable symbols to choose from. If no pair results in a valid encoding, then no auto-generated encoding following our encoding schema can be found. For the languages that were tested, a valid encoding was found as long as there were at least two usable symbols available to the token.

Consider the generated encoding in Listing 6 for the reduced HTML grammar. This encoding uses `A` as the escape symbol of the encoding and has encoded every keyword and keyword part to a seven letter character sequence. Since the encoding covers all keywords and keyword parts it can be considered a secure encoding for the language.

Having the encoding at hand raises the question where to apply it. Applying it to all nodes of a PT before unparsing would remove all keywords from the resulting message and hence render it invalid. Thus not all nodes must be encoded. This is the case because there are keywords that are used by the developer and are not an injection made by the attacker.

Within a parse tree every node is associated with a token type, namely the one that is used to parse that node. Whenever all nodes comply to their token type and neighbouring nodes do not contain keyword parts, unparsing such a PT does not require encoding to achieve a correct parsing round-trip. Since keywords are separate tokens with their own type, they cannot be found in a non-keyword node within a PT that does not need any encoding. Therefore only if a node does not match it's token type, i.e. results in two nodes when parsed, or contains the escape symbol, or a keyword part, the content needs to be encoded to make the content match the node's type again.

To search for such nodes in a PT, the *checker* is used. It checks these properties for every node in a PT. Thereby it detects injections that need to be encoded. For all nodes the checker reports, the encoding is applied.

The checker approach is especially useful in parser combinators, since it is possible to apply only the combined parser for that particular part of the language to a given node. Whereas when using state machine based parsers, parsing only a part of an input would require to introduce an accepting state to the state machine, leading to a more complex and error prone solution.

B. Applying Encoding and Decoding

The generated encoding is applied on a PT during unparsing through the encoder and applied in reverse by the decoder during the parsing of a message. The encoder is only called after the checker has found a token that needs to be encoded, whereas the decoder is called on every token. This is the case because the decoder cannot know which token has been encoded and which has not been encoded. This naturally leads to the fact that the escape symbol must not appear in the message prior to decoding. If it does, it makes the decoder create a keyword (part) that was not in the PT before unparsing. To further clarify this, consider the message `<p>ABBBBAB</p>` of the reduced HTML language. The parser creates three nodes within a PT from it, one for each keyword and one containing `ABBBBAB`. If the escape symbol `A` has not been encoded during unparsing, `ABBBBAB` is decoded to `<p>` leading to an injection caused by the decoder itself. If however, the escape symbol is encoded, the data `ABBBBAB` within the sending PT is encoded to `ABABABABBBBABABABAB` in the message. Thus after decoding it, it results again in `ABBBBAB`.

For the whole decoding process it is of the utmost importance that the encoding is performed only once per message. Otherwise the decoder cannot know how often it has to decode a given message. This would disrupt the bijectivity of the encoding, making it not uniquely decodable. Reconsidering the PT containing `<p>ABBBBAB</p>` before unparsing. Assuming the decoder decodes the message as often as it finds a string that matches an encoding. Then the input would be decoded to `<p><p></p>` which is obviously not the original content of

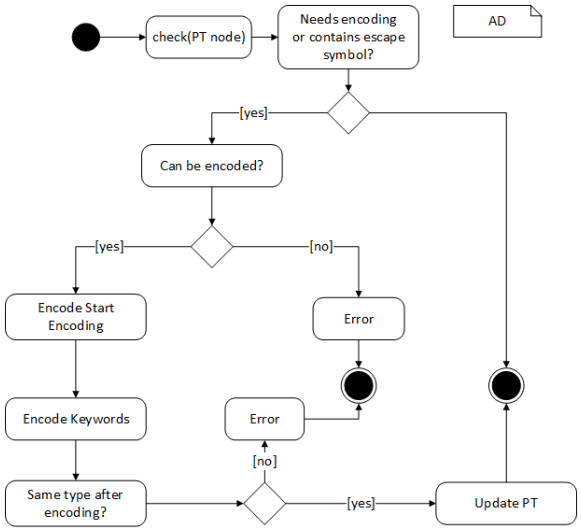

Fig. 1. Activity Diagram for encoding a PT node

the PT. Therefore encoding and decoding have to be applied exactly once.

Another pitfall is the ordering in which single encoding rules are applied during encoding. Simply replacing all occurrences of an encoded keyword, i.e. the escape symbol, with it's decoded equivalent might result in double replacements and hence altering of the original message. Therefore the encoding rules have to be applied in the same order when encoding and decoding a PT node. For decoding, McHammerCoder uses a sliding window of the size of the encoding and moves it over the content to be decoded. Whenever a decoding rule matches the contend in the sliding window it is applied and the window moves after that part of the message which has just been decoded. If nothing can be decoded within the window it is moved one character ahead. This is repeated until it reaches the end of the PT node. To clarify this, consider a PT node containing the input `<b>AA` before unparsing and encoding. Encoding it results in `ABBBBBAABABABBABABABAABABABA`. When decoding it without a sliding window it might be incorrectly decoded to `<bAB/p>`. Therefore using a sliding window for decoding is mandatory.

To sum up the encoding and decoding process in conjunction with the checker approach consider Figure 1. When encoding a PT node the first step is done by the checker – it takes the node and analyses its contents. If this node does not contain the escape symbol or any keyword (parts) and is of the type which is expected then it is left as is and the encoding process for this node is complete. In case this node needs to be encoded a check is performed to verify that an encoding has been generated beforehand for this token type. If no such encoding was generated the unparsing is aborted to prevent an injection. Otherwise, the encoder processes the node and does the actual encoding. The encoder always encodes the escape symbol first. Afterwards it encodes the keywords and keyword parts. After the whole node's content has been encoded a check is made to make sure that the token type has not been changed by the encoding. A change in the token type results in the abortion of the unparsing process. In the end only a fully encoded PT is unparsed.

Up to now only the encoding of symbols that lead to an invalid parsing round-trip has been taken into consideration. However, McHammerCoder also supports generating an encoding for characters that are not used by the language at all. This feature is called stuff-in stuff-out (SiSo) and can be activated or deactivated. Considering the token `Id` from the reduced HTML example in Listing 5 that allows all characters from `a` to `z`. Hence inserting `A` or `!` into the respective PT node would simply make the receiving parser reject that unparsed message. To support a correct parsing round-trip even for those cases, an encoding is generated for all symbols that are not used within the grammar.

Providing such a reliable parsing round-trip makes transmission of arbitrary data safe, even if the language definition does not consider such data at all. However, after parsing and decoding such data the PT contains data that is not defined within the grammar and therefore probably unexpected by the developer of the receiving program. This is contradictory to the LangSec approach of rejecting unexpected input within the parser and puts the burden of correctly handling such data again onto the developer.

VI. Evaluation

Since proving correctness of the generated (un)parser is out of scope of this work, we demonstrate the applicability and quality of the approach by implementing a language that uses all features of the (un)parser generator and testing the generated code with fuzzing.

First, to test the binary features a grammar for DNS requests was implemented and the PeachFuzzer [26] was used to create inputs for parsing. To enable Peach to generate DNS packets, a Peach Pit file was constructed. Fuzzing the parser with 60.000 valid and invalid DNS requests generated by Peach resulted in no crashes or hangs.

In addition a functional correctness evaluation was made to find if the parser rejects all invalid input and accepts all valid input. Therefore Bind DNS server was selected as a mature reference implementation of a DNS server and it's behaviour was compared to our generated parser. Based on the DNS response from Bind we determined if it's parser accepted the input. From the DNS requests generated by Peach approximately 20% were recognized as valid by Bind. For all generated DNS requests it was found that the generated parser accepts, respectively rejects, it if and only if Bind accepts, respectively rejects, it. Therefore for this test setup the generated parser recognizes the defined language.

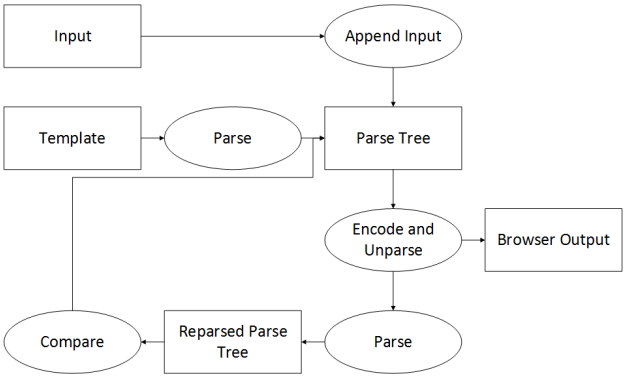

Fig. 2. Evaluation Setup

In order to evaluate the generated unparser and the derived encoding, a web application that uses input within different contexts of the produced HTML and JavaScript output was implemented. By choosing those languages, web application scanners can be levered as language specific fuzzers to identify errors in the encoding or unparsing process.

To follow the described approach, a grammar for those languages was implemented, the unparser and encoding were generated from it, and this unparser was used to create the application's output. Figure 2 shows the processing steps within the evaluation setup, that work as follows: First, a template which determines the static part of the generated website is parsed to initialize the PT. Then the input provided by the web application scanner is inserted to the PT. Next the updated PT is encoded and unparsed. Finally the resulting HTML document is sent to the web application scanner, that tries to determine if an injection attack was successful. Additionally, the document is parsed by the generated parser. After parsing and decoding, the resulting PT is compared with the PT used to generate the website. If a difference is found between both PTs – the unparser failed to encode some input, this is considered as an injection vulnerability.

ZAP [27], IBM AppScan [28], and FuzzDB [29] were used as web application scanners. ZAP's attacks were set on insane strength, the threshold for reporting vulnerabilities was set to low, however, it did not report any injections. IBM AppScan reported an *Microsoft Windows MHTML Cross-Site Scripting*. Since the web application is using XHTML, the browser will not interpret it as MHTML, which makes this finding a false positive. Lastly by using ZAP's fuzzer and the FuzzDB all attack strings from the sections: *XSS, XML* and *format-strings* were fed into the application. All attacks from the first two sections were prevented by correctly encoding them. Several attacks of the *format-strings* section were not encoded but still detected by the unparser, with SiSo mode disabled. When enabling SiSo mode those format-strings were processed and encoded.

Comparison of the unparsed PT with the one created from parsing the unparsed message did not show any differences for any of the performed injection attempts. Thus McHammerCoder successfully prevented all tested injections.

Concerning performance of the encoding process, the amount of generated encoding rules is the key performance factor. For our HTML and JavaScript grammar, which recognizes only a part of those languages, 21 rules were generated with SiSo mode disabled resp. 65485 when enabling it. Encoding with those 21 rules took 14.41 ms, while decoding added 0.17 ms when processing a 49 byte HTML message. With SiSo mode enabled the encoding of that message took 253.92 ms while decoding it took 158.58 ms.

VII. Conclusion

Preventing input based attacks requires developers to implement strict parsers and unparsers, that ensure a correct parsing round-trip from PT serialized data and back. Ad-Hoc solutions to this problem often contain security vulnerabilities like arbitrary code execution and injections. Solving these vulnerabilities for all deterministic context-free languages requires an approach where the (un)parser implementation is reused, i.e. derived from a grammar. This makes language and encoding definition within the grammar vital for preventing input-based vulnerabilities.

To free language developers from considering all cases where input needs to be encoded and hence ease language definition, an approach that derives an encoding from a given context-free grammar was presented. By incorporating this encoding with the binary-capable Hammer parser, a comprehensive solution for a correct parsing round-trip is provided. This was achieved by extending the MontiCore grammar language with binary elements, and generating a Hammer parser as well as an unparser from it. The generated (un)parser along with the encoder and decoder were evaluated by implementing a binary DNS query grammar and an HTML grammar. Fuzzing the generated (un)parsers where neither an arbitrary code execution nor an injection was found. The presented McHammerCoder approach allows developers to prevent input-based attacks on all context-free (binary) languages by simply defining the language in a grammar.

Clearly, defining communication protocols using grammars is uncommon for developers. Considering other representations of grammars that developers are more familiar with might improve acceptance and use of the proposed approach. Another challenge where the approach still needs to be applied are parser differentials. Solving it would provide interoperability of the encoding with existing parsers that interpret messages differently then the McHammerCoder parser. From an offensive security perspective, extracting the grammar from an already implemented (un)parser might be interesting to identify edge cases that might turn out to be injection vulnerabilities.

REFERENCES

[1] S. Bratus, T. Darley, M. Locasto, M. L. Patterson, R. Shapiro, and A. Shubina, "Beyond planted bugs in "trusting trust": The input-processing frontier," *Security Privacy, IEEE*, vol. 12, no. 1, pp. 83–87, Jan 2014.

[2] A. V. Aho, *Compilers: Principles, Techniques and Tools*. Pearson Education India, 2003.

[3] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[4] W. H. Burge, *Recursive programming techniques*, ser. Systems programming series. Reading, MA: Addison-Wesley, 1975.

[5] G. Hutton, "Higher-order functions for parsing," *Journal of Functional Programming*, vol. 2, no. 3, pp. 323–343, 007 1992.

[6] J. Fokker, *Functional parsers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–23.

[7] J. Bangert and N. Zeldovich, "Nail: A practical tool for parsing and generating data formats," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 615–628.

[8] M. L. Patterson, "Hammer," https://github.com/UpstandingHackers/hammer, 2016.

[9] M. van den Brand and E. Visser, "Generation of Formatters for Context-free Languages," *ACM Transactions Software Engineering Methodology (TOSEM)*, vol. 5, no. 1, pp. 1–41, January 1996.

[10] K. Matsuda and M. Wang, "FliPpr: A Prettier Invertible Printing System," in *Programming Languages and Systems*, ser. LNCS, M. Felleisen and P. Gardner, Eds. Springer, 2013, vol. 7792, pp. 101–120.

[11] L. Hermerschmidt, S. Kugelmann, and B. Rumpe, "Towards more security in data exchange: Defining unparsers with context-sensitive encoders for context-free grammars," in *2015 IEEE Security and Privacy Workshops*, May 2015, pp. 134–141.

[12] J. Hughes, "The design of a pretty-printing library," in *Advanced Functional Programming*, ser. LNCS, J. Jeuring and E. Meijer, Eds. Springer, 1995, vol. 925, pp. 53–96.

[13] P. Wadler, "A Prettier Printer," in *Journal of Functional Programming*. Palgrave Macmillan, 1998, pp. 223–244.

[14] V. Zaytsev and A. H. Bagge, "Parsing in a Broad Sense," in *Model-Driven Engineering Languages and Systems (MODELS)*, ser. LNCS, J. Dingel, W. Schulte, I. Ramos, S. Abraho, and E. Insfran, Eds. Springer, 2014, vol. 8767, pp. 50–67.

[15] T. Rendel and K. Ostermann, "Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing," in *ACM Haskell Symposium on Haskell*. ACM, 2010, pp. 1–12.

[16] L. Sassaman, M. L. Patterson, S. Bratus, and A. Shubina, "The halting problems of network stack insecurity," *login*, vol. 36, no. 6, pp. 22–32, 2011.

[17] H. Krahn, B. Rumpe, and S. Völkel, "Monticore: a framework for compositional development of domain specific languages," *International journal on software tools for technology transfer*, vol. 12, no. 5, pp. 353–372, 2010.

[18] D. Ghosh, *DSLs in Action*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2010.

[19] J. Williams, N. Mattatall, and J. Manico, "XSS (cross site scripting) prevention cheat sheet," https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet, 2016.

[20] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A systematic analysis of XSS sanitization in web application frameworks," in *Computer Security–ESORICS 2011*. Springer, 2011, pp. 150–171.

[21] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: a client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 2006, pp. 330–337.

[22] A. Dabirsiaghi, "ro0, a Java agent that protects applications from deserialization attacks," https://github.com/Contrast-Security-OSS/contrast-rO0, 2016.

[23] A. Muñoz, "The perils of java deserialization," in *HPE Security Research*. Hewlett Packard Enterprise, 2016.

[24] F. Momot, S. Bratus, S. Hallberg, and M. L. Patterson, "The seven turrets of babel: A taxonomy of langsec errors and how to expunge them," in *IEEE SecDev*, 2016.

[25] D. Long and W. Jia, "The most efficient uniquely decipherable encoding schemes," in *Web Information Systems Engineering, 2000. Proceedings of the First International Conference on*, vol. 1. IEEE, 2000, pp. 159–163.

[26] M. Eddington, "Peach fuzzing platform," *Peach Fuzzer*, p. 34, 2011.

[27] S. Bennetts and O. Community, "Zed Attack Proxy," https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project, 2017.

[28] IBM, "Security AppScan," http://www-03.ibm.com/software/products/en/appscan, 2017.

[29] A. Muntner, "FuzzDB," https://github.com/fuzzdb-project/fuzzdb, 2017.