Mining Input Grammars from Dynamic Taints

Matthias Höschele
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
hoeschele@cs.uni-saarland.de

Andreas Zeller
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
zeller@cs.uni-saarland.de

ABSTRACT

Knowing which part of a program processes which parts of an input can reveal the structure of the input as well as the structure of the program. In a URL `http://www.example.com/path/`, for instance, the protocol `http`, the host `www.example.com`, and the path `path` would be handled by different functions and stored in different variables. Given a set of sample inputs, we use *dynamic tainting* to trace the data flow of each input character, and aggregate those input fragments that would be handled by the same function into lexical and syntactical entities. The result is a *context-free grammar* that reflects valid input structure. In its evaluation, our AUTOGRAM prototype automatically produced readable and structurally accurate grammars for inputs like URLs, spreadsheets or configuration files. The resulting grammars not only allow simple reverse engineering of input formats, but can also directly serve as input for test generators.

CCS Concepts

•**Software and its engineering** → **Input / output;** *Dynamic analysis;* •**Theory of computation** → *Grammars and context-free languages;* •**Social and professional topics** → Software reverse engineering; •**Applied computing** → Document analysis;

Keywords

Input formats; context-free grammars; dynamic tainting; fuzzing

1. INTRODUCTION

Since the invention of the Turing machine, a program is typically described as a machine that input and output a *string* of symbols. The set of such strings that the machine accepts or produces is called a *language*. The field of formal language theory long has studied the structural aspects of such languages, using *formal languages* like regular expressions or context-free grammars to exactly specify the language. The practical importance of such formal languages cannot be overstated. In programming languages, software systems, computer networks, or general software development, formal languages (and equivalent automata diagrams) are among the most frequently used methods to specify inputs and outputs, and consequently, regular expressions and grammars are an essential part of computer science curricula.

```
http://user:password@www.google.com:80/command?
    foo=bar&lorem=ipsum#fragment
http://www.guardian.co.uk/sports/worldcup#results
ftp://bob:12345@ftp.example.com/oss/debian7.iso
```

Figure 1: Sample URL inputs

```
URL ::= PROTOCOL '://' AUTHORITY PATH
        ['?' QUERY] ['#' REF]
AUTHORITY ::= [USERINFO '@'] HOST [':' PORT]
PROTOCOL ::= 'http' | 'ftp'
USERINFO ::= /[a-z]+/ ':' /[a-z]+/
HOST ::= /[a-z.]+/
PORT ::= '80'
PATH ::= /\/[a-z0-9.]*/
QUERY ::= 'foo=bar&lorem=ipsum'
REF ::= /[a-z]+/
```

Figure 2: Grammar derived by AUTOGRAM from `java.net.URL` processing the inputs in Figure 1. Optional parts are enclosed in brackets `[...]`; regular expression shorthands are enclosed in `/.../`.

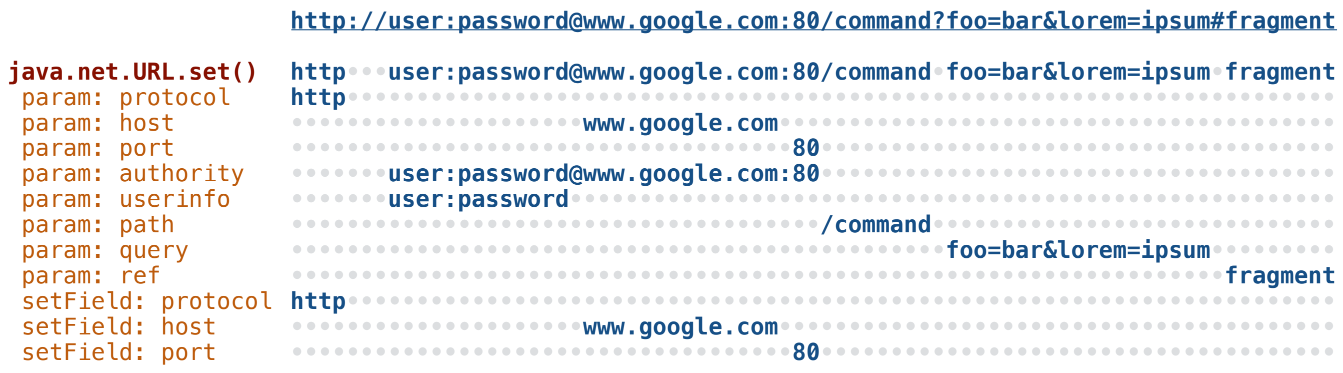
In this paper, we present a novel practical method that, given a set of program runs with inputs, automatically produces a *context-free grammar* that represents the language of the inputs seen. The resulting grammar facilitates understanding of the input structure; can serve as a base for automated test generation by feeding it into a producer; and can be used by a computer to parse, decompose, and analyze other inputs.

Here is an example. `java.net.URL` is a Java class that parses a Uniform Resource Locator (URL) into its constituents. Assume a program *p* that uses `java.net.URL` to parse the URLs given in Figure 1. Given the program *p* and these inputs, our AUTOGRAM prototype automatically produces the grammar shown in Figure 2, which pretty accurately reflects the structure of the URLs processed.

How do we obtain this grammar? The key idea is to *dynamically observe how input is processed in a program.* We instrument the program with *dynamic taints* that during execution, tagging each piece of data with the *input fragment it comes from.* Now, if some function of the program processes only a part of the input, or if a part or a value derived from it is stored in a variable, then this part becomes a *syntactical entity*.

In our example, the method `java.net.URL.set()` eventually stores the URL components as parsed in the `java.net.URL` class. Figure 3 shows the taints from the original input as its parts are being passed as arguments to `java.net.URL.set()`. The

ASE 2016 Singapore
© 2016 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4



```
                    http://user:password@www.google.com:80/command?foo=bar&lorem=ipsum#fragment

java.net.URL.set()  http   user:password@www.google.com:80/command foo=bar&lorem=ipsum fragment
 param: protocol    http
 param: host                             www.google.com
 param: port                                            80
 param: authority          user:password@www.google.com:80
 param: userinfo           user:password
 param: path                                              /command
 param: query                                                      foo=bar&lorem=ipsum
 param: ref                                                                            fragment
 setField: protocol http
 setField: host                          www.google.com
 setField: port                                         80
```

Figure 3: Dynamic tainting in AUTOGRAM. At the end of URL parsing, `java.net.URL.set()` is invoked with parameters such as `protocol = "http"`, `host = "www.google.com"`, or `port = 80`. AUTOGRAM instruments the program such that each input character is associated with its position in the input. This *taint* propagates to derived values and variables as the input is processed: As `java.net.URL.set()` stores its arguments in object attributes, their taints propagate along.

`PROTOCOL` part of the URL (`"http"` and `"ftp"` in our inputs) is passed as argument for the `protocol` parameter (and later stored in an object attribute of the same name). This both identifies the `PROTOCOL` part and gives the entity a readable and meaningful name. The arguments `host` and `port` thus also lead to entities in the grammar.

If a part processed by a function subsumes smaller parts, we introduce a rule that composes the part out of its subparts. This is how the `AUTHORITY` entity is composed of smaller parts such as `HOST` or the optional `USERINFO` or `PORT`; see in Figure 3 how the `authority` argument is just an aggregation of its smaller parts. Not all entities decompose into smaller fragments, though; since `java.net.URL` never stores or handles user and password separately, we retain a single entity `userinfo`; likewise, queries or hostnames would not be further decomposed.

After assigning entities and their hierarchy, we use a *regular expression learner* to generalize multiple terminals into matching automata; thus, the `HOST` entity becomes a string of lowercase letters and dots. Our approach thus results in grammars as shown in Figure 2—grammars with significant impact in software development:

1. The grammar gives humans immediate and detailed insights into the structure of inputs, thereby facilitating *reverse engineering* of input formats as well as manually writing *valid test inputs.* To value this contribution, simply consider Figure 2 and assess how such knowledge, automatically produced, could facilitate reverse engineering.

2. The grammar can immediately be used by *test generators* to produce high numbers of varied and valid inputs, thus facilitating automated robustness testing and fuzzing. While the grammars produced may *overgeneralize* (a real host name, for instance, may not contain two consecutive dots, which is not reflected in our grammar), this is not a major concern in test generation, as invalid inputs would be rejected by the program—just as a compiler may reject syntactically valid, but semantically invalid programs.

3. The grammar vastly simplifies the creation of *parsing programs* that decompose existing inputs into their constituents. Note, though, that our grammar may *overspecialize,* and may require human inspection and/or adjustment. For instance, in Figure 2, the `PORT` and `QUERY` entities, for instance, reflect the respective single values seen so far; these would be generalized either by providing additional sample inputs, or by having a human adjust the respective rules.

The contributions of this paper are as follows. To the best of our knowledge, the present work is the first to *automatically derive a context-free input structure from given inputs,* making use of a *provided processing program* to derive lexical and syntactical structure as well as entity names. This is in contrast to related work (Section 2), which does not make use of existing programs, and therefore has to rely on additional lexical or structural hints.

Minor contributions include our method to associate processed data with positions in the program input (Section 3) as well as our grammar synthesis from jointly processed entities, including deriving entity names from variable and function identifiers (Section 4). Section 5 evaluates our AUTOGRAM prototype in terms of accuracy and completeness, where we use it to parse and produce yet unseen inputs, respectively, on a set of common input formats. We close with conclusion and future work in Section 6.

2. BACKGROUND

AUTOGRAM joins three areas: language induction, tracking input origins, and data tainting.

Language Induction. Languages play a crucial role in software, both in programs as in data. The Wikipedia page on file formats lists more than 1,300 commonly used file formats—which is likely only a small fraction of the many proprietary data formats used by programs all across the world. Given these many languages, it is only natural that researchers have thought about how to reverse engineer languages from given inputs. The field of *language induction* has provided a multitude of approaches to infer languages and patterns from (mostly natural language) text; for a comprehensive overview, we recommend the textbook by de la Higuera [4]. The vast majority of approaches, however, focuses on *regular languages.* The single exception is the work by Sakakibara [5, 6], which infers context-free grammars from given strings if given “skeletons”, i.e. some form of structural hints. By leveraging program executions, AUTOGRAM is able to construct such hints and decompose the input into its structure.

Input Origins. To the best of our knowledge, relating input fragments to code fragments that process it was first suggested by Clause and Orso for their PENUMBRA work [3]. Their idea is to track inputs as they propagate during execution, and if some function fails, they would be able to identify those inputs that the function uses and therefore would be responsible for the failure. PENUMBRA computes such origins only for specific variables, though; whereas AUTOGRAM tracks the *entire* input as it is being processed.

Data Tainting. AUTOGRAM relies on *dynamic tainting* to identify the origins of data fragments and track their flow during program execution. Dynamic tainting allows us to precisely identify which parts of a programs input are read, stored and processed at any point in time.
The dynamic tainting as implemented in AUTOGRAM is inspired by Phosphor [1], a portable implementation of dynamic taint analysis based on bytecode instrumentation of all classes including the Java API. In contrast to Phosphor, however, AUTOGRAM can taint arbitrary values (including primitives and internalized strings) with string origin descriptions, which consists of arbitrary long lists of input regions.

All in all, we are not aware of an approach that would use a mapping between input and locations processing them to derive structural input descriptions, let alone context-free grammars, as with AUTOGRAM.

3. TAINTING INPUT CHARACTERS

The tainting framework of AUTOGRAM is inspired by the Phosphor [1] work. It is based on bytecode instrumentation that in constrast to Phosphor does not modify method signatures and uses separate shadow memory to store taint information for local variables, stack values and arrays. This separation and the usage of an unmodified version of ASM [2] make it easier to combine it with additional instrumentation and to make it interoperable with uninstrumented code. Our implementation can also add shadow information to references. This special feature enables us to handle features like Java `String` internalization that returns a canonical reference for equal `String` objects. The current implementation supports multiple taint sources based on system resources like files or sockets and represents the taint information for each taint source as bit sets. The implementation is currently restricted to single threaded programs and leaves a lot of room for performance optimization which was not the focus of this work. Since the Java HotSpot VM does not permit the modification of certain core classes using a Java agent, the Java API needs to be instrumented offline and included as bootstrap classes. All other instrumentation is applied by a Java agent. The agent supports caching of previously instrumented classes to avoid unnecessary overhead for repeated executions.

While running a program we use our taint analysis framework to create a trace of events for each relevant program point:

Method Entry. Each method entry is logged, including the taint information for all argument values.

Method Exit. Each method exit is logged, including the taint information of the return value.

Field Access. Each field access is logged with its access type (read-/write) and the taint information of the value that was read-/written.

Array Access. Each array access is logged with its access type (read/write) and the taint information of the value that was read/written.

The tracer uses a binary format that uses a compact representation of the program points based on unique ids that are assigned to classes, methods and fields during instrumentation. Executions with millions of calls therefore result in traces of a few Megabytes. Since our tainting framework is not yet optimized, traced programs run approximately 100 times slower.

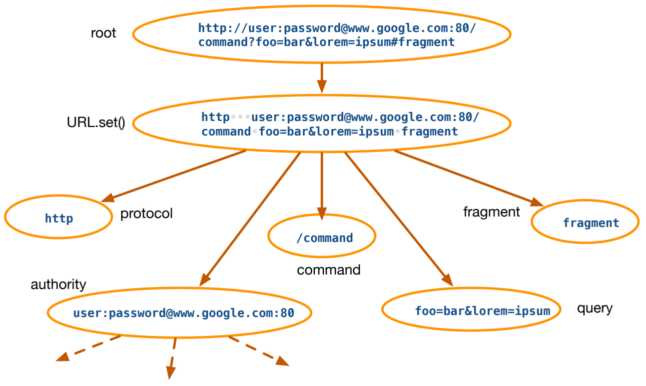

root
http://user:password@www.google.com:80/command?foo=bar&lorem=ipsum#fragment
URL.set()
http user:password@www.google.com:80/command foo=bar&lorem=ipsum fragment
http
protocol
/command
command
fragment
fragment
authority
user:password@www.google.com:80
foo=bar&lorem=ipsum
query

Figure 4: Interval tree for the interval set from Figure 1

The resulting event traces allow us to reconstruct the dynamic call tree as well as the input fragments that were processed during a method call. Input fragments processed by callees as well as field and array accesses are attributed to the initiating method. To extract this information from the trace, we sequentially read the events from the trace, and merge the input fragments of callees to the caller on encountering the exit event of the callee.

4. SYNTHESIZING GRAMMARS

4.1 Identifying Entities

We now describe how we identify individual entities to produce a grammar. Our method works along the following steps, each illustrated by the URL example in Figure 1 and Figure 2.

Intervals. Since parsing breaks down the input to fragments of decreasing size, this can be observed during program execution. Induced by the dynamic call tree and the corresponding taint information, we can derive the set of intervals $I \subseteq \{[i,j] | i,j \in \{0,\ldots,n-1\}, i \leq j\}$ where n is the number of characters of the input, such that for each interval $i \in I$ there is either

- a *function* that processes the input fragment corresponding to the interval i, or
- a *variable* or *parameter* that stores the same input fragment, or
- the input fragment or a value derived from it is *returned* by a function.

Figure 1 shows the interval set I for our URL example. When calling `java.net.URL.set()`, the `protocol` parameter is associated with the leading `"http"` string, or the interval $i = [1,4]$ of the original input. The `URL.set()` function processes all values given as parameters, and thus is associated with the intervals $i_1 = [1,4]$ `"http"`, $i_2 = [8,44]$ `"user.../command"`, etc., as depicted in Figure 1.

Interval Trees. This set I of intervals also induces a *tree* T' *of intervals* where the root $r = [0, n-1]$ and $\forall i,j \in I$, j is a child of i if and only if $j \subset i$ and $\neg\exists k \in I, j \subset k$. In Figure 1, the root r would be the entire input (shown on top); its child would be the interval of `java.net.URL.set()`, since it is no subset of another interval. The intervals of the `protocol`, `authority`, `path`, `query`, and `ref` parameters would be the next direct children (Figure 4).

Computing Interval Trees. The interval tree T' can be computed by applying *depth first search* to the dynamic call tree and *merging* the sets of processed input fragments of all children

on exiting a node. Our goal is to derive a interval tree free of conflicting overlaps T such that for all nodes $i, j \in T \subseteq I, i \cap j \neq \emptyset \iff i \subset j \vee j \subset i$. In the following, we call such trees *pure* interval trees.
We assume the tree T corresponds to a parse tree such that the leaves correspond to terminal symbols, and inner nodes to productions of nonterminal symbols. T' does not necessarily satisfy the additional constraints given for T. Parser implementations that make use of lookahead for instance will usually result in nodes with overlapping children.

From Intervals to Entities. We associate each interval with the set of corresponding elements in the dynamic call tree (i.e., method invocations, parameters, return values, field acesses). We can thus check if nodes are compatible and can be used to derive productions for for the same nonterminal symbol.

Deriving Production Rules. To derive production rules we look at all sets N of compatible nodes in the set of interval trees that we computed from our samples. The children of each node correspond to a sequence of terminal and nonterminal symbols that can be a right-hand side of a production rule for the nonterminal symbol derived from N.

4.2 Lookahead and Terminals

Most parsers require some form of *lookahead* to deterministically decide language membership and derive a parse tree based on the next symbol to parse. As an example, consider a JSON parser, where `JsonParser.readStringInternal()` reads a string including the quotation marks and a line break character, as in `"foo"¶`. The contained string is finally consumed in a second method `JsonParser.endCapture()` including the terminating quotation mark due to *lookahead*, as in `foo"`. Which method should the final quotation mark be attributed to?

To resolve possible ambiguities, we apply a simple heuristic that assumes left to right processing of the input. For each node $n \in I$ with children $i = [i_l, i_r], j = [j_l, j_r] \in I$ such that $i_l < j_l$, we derive a replacement interval $i' = [i_l, j_l - 1]$. We also recursively remove all children $c = [c_l, c_r]$ of i' if $c \subseteq j$ or replace them with intervals $c' = [c_l, j_l - 1]$.

In our case, we attribute the terminating quotation mark to the method `JsonParser.readStringInternal()`, which also processes the mark after calling `JsonParser.endCapture()`. We can thus remove it from the `JsonParser.endCapture()` interval corresponding to `foo"` to obtain the replacement that represents only the encompassed string `foo`.

4.3 Readable Entity Names

Since we are working with Java bytecode we do not have to resort to derive random names for nonterminal symbols. Method and field names can always be extracted from classes and provide good candidates for descriptive symbol names. If the bytecode includes debug symbols our approach is also able to derive symbol names from method parameter names. AUTOGRAM implements simple heuristics that define a order in which names of functions (including constructors), parameters, and fields are considered.

Certain names like `getChar` or similar common functions can also be ignored since they will be called for most characters and offer limited help in finding descriptive symbol names. As further improvement, we implemented simple heuristics for simplifying candidate names by discarding common prefixes to names like `parse`, `read`, `skip`, `get`, or `set`.

5. EVALUATION

5.1 Evaluation Setup

Now that we have described our approach, let us see how well it works. We apply AUTOGRAM on a number of example programs, learn their input grammars, and evaluate each grammar for accuracy and completeness.

5.1.1 Accuracy

Our first research question concerns the *accuracy* of the generated grammars: To which extent does the grammar *overgeneralize*—that is, does it represent strings that would be rejected by the programs at hand? To this end, we use the AUTOGRAM-produced grammars as *producers*—that is, we start with the start symbol and continuously expand nonterminals according to grammar rules.

The strings resulting from such production would then be fed into the program in question, which would either accept or reject it. Any rejection is a sign of overgeneralization: The string `"http://xyzzy.:80quux/"` could be produced by Figure 2, but would be rejected by the Java URL parser, since the path does not begin with a `'/'` character.

For each production that requires a choice we pick alternatives randomly (e.g. choosing between alternatives, number of repetitions). For optional parts we also randomly choose to expand them, however we additionally enforce a maximum depth at which we prevent expansion of optional parts to ensure termination.

5.1.2 Completeness

Our second research question concerns the *completeness* of the generated grammars: To which extent does the grammar *overspecify*—that is, not contain strings that actually would be accepted by the programs in question? To this end, we manually produced *reference grammars* for each test subject that would serve as ground truth for the respective input language.

We then used the reference grammars as *producers* that would create arbitrary strings; these would then be parsed by the grammars produced by AUTOGRAM. (To this end, we adapted the grammars produced by AUTOGRAM into a semantically equivalent LL(1) form that could be directly used by a parser generator). If a AUTOGRAM grammar would reject a string produced by the reference grammar, this would have been a sign of the AUTOGRAM grammar being too specific.

5.2 Subjects and Results

Table 1 summarizes our evaluation subjects. Our subjects include parts of the Java Standard API that are used to process URLs and property files. The other subjects are open source projects that implement support for CSV, INI and JSON formats. For each subject, we would use the reference grammar to produce 1,000 sample inputs, which then would be fed into the program to produce a AUTOGRAM grammar. The resulting grammar would then be subject to accuracy and completeness evaluation, as described above. Each subject is described in a separate section.

5.2.1 URLs

In Figure 5, we show the AUTOGRAM grammar obtained from the `java.net.URL` class after feeding the parser with 1,000 random samples. This grammar is more general than the one in Figure 2 that was obtained from only the three samples listed in Figure 1.

[1]https://commons.apache.org/proper/commons-csv/
[2]http://ini4j.sourceforge.net/
[3]https://github.com/ralfstx/minimal-json

Table 1: Test Subjects

Subject	Data Format	Format Purpose
`java.lang.URL`	URL	Uniform Resource Locators; used as Web addresses
Apache Commons CSV[1]	CSV	Comma-separated values; used in spreadsheets
INI4J[2]	INI	Configuration files consisting of sections with key/value pairs
Minimal JSON[3]	JSON	Human-readable text to transmit nested data objects with attributes and values

Note that the generalization occurs much more at the lexical rather than at the syntactical level; an effect we have frequently observed in our subjects.

In our experiment, the AUTOGRAM grammar accepted *all* inputs produced by the reference grammar; it is thus 100% complete. However, it does overgeneralize and thus is not entirely accurate: Of 10,000 strings it produces, 1,772 are rejected by the `java.net.URL` parser; all because of the URL containing a port number, but the path not starting with `'/'`—the same overgeneralization as already detailed in Section 5.1.1. We thus compute its accuracy as $(10,000 - 1,772)/10,000 = 8,228/10,000 = 82.3\%$.

5.2.2 *Comma-Separated Values*

Our next subject is the simplest in terms of grammars, namely comma-separated values as processed by Apache Commons CSV. Comma-separated values are frequently used as data exchange format, notably between spreadsheet applications.

The AUTOGRAM grammar inferred from Apache Commons CSV is shown in Figure 6; we see that a CSV file consists of sequences of record values (strings) separated by either commas or newlines. Since Apache Commons CSV performs no further processing of the strings read, AUTOGRAM leaves them as general as they are. Of note is that in the grammar, a `TOKEN` is preceded by a newline or a comma; a human-written grammar would make these separators between tokens and place them *after* a token and assign the newline symbol to records instead of tokens. In terms of accuracy and completeness, the AUTOGRAM grammar scores 100% in both.

5.2.3 *INI Configuration Files*

INI files have a similar key/value format as *Java property files*; they are mostly used on the Windows platform to store configuration parameters. Applied on the INI4J framework, AUTOGRAM returns the grammar in Figure 7. We see that on top of key/-value options, INI files also have section titles in square brackets (`SECTIONLINE`) as well as comments (`COMMENT`) starting with semicolons `';'`.

Of special note is the `LINESKIPCOMMENT` line, which is unnecessarily complex due to whitespace processing. In our future work (Section 6), we want to derive general lexical rules such as all whitespace between items being skipped, making the grammar both more general and simpler. The learned grammar does not describe an INI file as a sequence of sections, though, and thus allows options at the beginning of a file without defining a section first. This is due to reading options, sections and comments in a single loop, making the grammar only 64.6% complete.

```
URL ::= PROTOCOL '://' AUTHORITY
     [PATH] ['?' QUERY] ['#' REF]
PROTOCOL ::= 'http' | 'ftp'
AUTHORITY ::= [USERINFO '@'] HOST [':' PORT]
USERINFO ::= /[a-zA-Z0-9:]+/
HOST ::= /[a-zA-Z0-9.]/
PORT ::= /[0-9]+/
PATH ::= /[a-zA-Z0-9.,\/]+/
QUERY ::= /[a-zA-Z0-9&=]+/
REF ::= /[a-zA-Z0-9]+/
```

Figure 5: Generalized URL grammar derived by AUTOGRAM

```
CSV ::= RECORD+
RECORD ::= TOKEN+
TOKEN ::= /[\n,]*/ RECORDVALUE
RECORDVALUE ::= /[a-zA-Z0-9.;\/ ]+/
```

Figure 6: CSV grammar derived by AUTOGRAM.

```
INI ::= LINESKIPCOMMENT+
LINESKIPCOMMENT ::=
   [(/[ \t]*/
       (COMMENT | SECTIONLINE | OPTION)
     /[ \t]*/ '\n')+]
   /[ \t]*/
     (COMMENT | SECTIONLINE | OPTION)
   /[ \t]*/ ['\n']
COMMENT ::= ';' /[a-zA-Z0-9_ ]+/
OPTION ::= KEY /[ \t]*/ '=' /[ \t]*/ VALUE
KEY ::= /[a-zA-Z0-9_]+/
VALUE ::= /[a-zA-Z0-9_ ]+/
SECTIONLINE ::=
    '[' /[ \t]*/ SECTION /[ \t]*/ ']'
SECTION ::= /[a-zA-Z0-9 ]+/
```

Figure 7: INI configuration grammar derived by AUTOGRAM.

5.2.4 *JSON Objects*

Strictly speaking, the grammars of all our past examples could also have been expressed by an (albeit pretty complex) regular expression. That is because they do not contain recursive structures, which can only be properly expressed with context-free grammars. JSON (JavaScript Object Notation) is a data interchange format for primitive and structured objects commonly used on JavaScript platforms, but available for several languages. The *Minimal JSON* library provides central JSON parsing and processing capabilities for Java programs.

In Figure 8, we see the grammar as inferred from AUTOGRAM through dynamic analysis of *Minimal JSON*. We can clearly distinguish the individual value types, from primitive types (`TRUE`, `FALSE`, `NUMBER`, ...) to recursively structured types (`ARRAY`, `JSONOBJECT`). Of special note is `JSONOBJECT`, in which a human grammar designer would probably factor out key/value pairs into a separate entity, and thus simplify the grammar. However, the rule reflects the way *Minimal JSON* processes JSON objects, namely in a single loop, appending key/value attributes to an object as it sees them. In our future work, we will borrow from existing language learners to identify structural repetitions and factor these out into (anonymous) entities. In our experiments, the JSON grammar proved to be 100% accurate and complete.

```
JSON ::= WHITESPACE VALUE WHITESPACE
WHITESPACE ::= /[ \n\t]+/
VALUE ::= JSONOBJECT | ARRAY | STRINGVALUE |
          TRUE | FALSE | NULL | NUMBER
TRUE ::= 'true'
FALSE ::= 'false'
NULL ::= 'null'
NUMBER ::= ['-'] /[0-9]+/
STRINGVALUE ::= '"' INTERNALSTRING '"'
INTERNALSTRING ::= /[a-zA-Z0-9 ]+/
ARRAY ::=
'['
  [WHITESPACE]
  [VALUE [WHITESPACE]
  [',' [WHITESPACE] VALUE [WHITESPACE]]+]
']'
JSONOBJECT ::=
'{'
  [WHITESPACE]
  [STRINGVALUE [WHITESPACE] ':' [WHITESPACE]
   VALUE [WHITESPACE]
  [',' [WHITESPACE]
   STRINGVALUE [WHITESPACE] ':' [WHITESPACE]
      VALUE [WHITESPACE]]
  +]
'}'
```

Figure 8: JSON grammar derived by AUTOGRAM.

Table 2: Evaluation Results

Subject	Accuracy	Completeness
`java.lang.URL`	82.3%	100.0%
Apache Commons CSV	100.0%	100.0%
INI4J	64.6%	100.0%
Minimal JSON	100.0%	100.0%

5.3 Evaluation Summary

All our results regarding accuracy and completeness are summarized in Table 2. Altogether, the grammars produced by AUTOGRAM proved to be 100% complete: They accepted all strings produced by the reference parsers. This indicates a high usefulness of the resulting grammars for reverse engineering purposes.

In our evaluation setting, grammars inferred by AUTOGRAM all accepted 100% of syntactically legal inputs.

On the other hand, not all parsers were 100% accurate, indicating that it may be necessary for humans to make adjustments before the grammars would produce syntactically valid outputs only. However, the main purpose of such outputs would typically be robusteness testing; and if a certain percentage of inputs would be filtered out as invalid by the processing program, that would only hurt performance, but not functionality.

In our evaluation setting, most grammars inferred by AUTOGRAM produced 100% syntactically legal outputs.

Note, though, that there also are theoretical limits to what a grammar can achieve. While it would be desirable to extract grammars that would be both 100% complete and accurate, a grammar may have to be *Turing complete* to achieve these levels. Descriptions of what makes a valid program, for instance, are not only governed by the program syntax, but also by semantical rules that can only be expressed in a Turing complete formalism—which typically would be the source code of the compiler or interpreter. As of now, AUTOGRAM only aims for extracting syntactical structure; and we are already very satisfied with its results.

5.4 Threats to Validity

Our evaluation is subject to several threats to validity. The most important is *external validity.* While we have evaluated AUTOGRAM on a small set of parsers for commonly used file formats, we make no claim that these results would generalize; rather, they are to be seen as indication for the potential usefulness of an approach such as AUTOGRAM. We see several opportunities for improvement and future research; while we regard AUTOGRAM as an achievement, it is but the first step into an exciting and wide field of program-supported grammar inference.

In our evaluation, we trained all parsing subjects from randomly generated samples. These may or may not reflect the variability seen in real-world samples and thus cannot be seen as representative. Hence, training AUTOGRAM from real-world samples is likely to result in different accuracy and completeness measures, in particular when the training set is small.

6. CONCLUSION

The AUTOGRAM approach provides a simple means to reverse engineer the structure of program inputs to context-free grammars, by associating parts of a program with the parts of the input that they process. The potential of this approach are clear, ranging from better understanding of inputs over better testing to creation of parsing programs, and generally opening a new field of program-supported grammar inference. Our main future work will be using the inferred grammars for systematic fuzz testing, bypassing lexical and syntactical checks to deep into the program.
AUTOGRAM and all benchmarks presented in this work are available for evaluation purposes. For details, see our Web site:

https://www.st.cs.uni-saarland.de/models/autogram/

Acknowledgments. We thank the anonymous reviewers for their constructive comments. This work is supported by the European Research Council, Grant "SPECMATE".

7. REFERENCES

[1] J. Bell and G. Kaiser. Phosphor: Illuminating dynamic data flow in commodity JVMs. In *ACM SIGPLAN Notices*, volume 49, pages 83–101. ACM, 2014.

[2] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30:19, 2002.

[3] J. Clause and A. Orso. Penumbra: automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 249–260. ACM, 2009.

[4] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA, 2010.

[5] Y. Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23–60, 1992.

[6] Y. Sakakibara. Learning context-free grammars using tabular representations. *Pattern Recogn.*, 38(9):1372–1383, Sept. 2005.